TURING 图灵程序设计丛书

Packt>

Advanced Machine Learning with Python

Python 高级机器学习

[英] 约翰·哈蒂 著　王大鹏　管晨　郭鹏程 译

■ 用Python语言详细解读前沿机器学习技术，助你高效解决数据科学难题



中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

约翰 · 哈蒂
（John Hearty）

数据科学家，数据科学和架构工程领域咨询专家，曾为微软、世嘉等公司提供咨询服务，致力于将机器学习前沿技术付诸实践，热衷于用Python构建机器学习解决方案。现任职于NuData Security公司。

王大鹏

数据科学从业者，人工智能爱好者。毕业于山东财经大学金融数学系，现从事智能教育、程序设计和算法应用等方面的工作，对数据挖掘和机器学习有一定研究。

管晨

毕业于山东财经大学金融数学系，在数据分析和数据可视化领域拥有丰富经验，现从事商业地产和教育方面的商业智能工作，爱好语言与翻译。

郭鹏程

物理专业出身，天文学博士、科普爱好者、数据科学家，致力于将数据科学应用于实际业务，从事数据科学相关的培训、咨询、设计和开发工作。目前任教于山东财经大学数学与数量经济学院。

数字版权声明

TURING 图灵程序设计丛书

Advanced Machine Learning with Python

Python高级机器学习

[英] 约翰·哈蒂　著
王大鹏　管　晨　郭鹏程　译

人民邮电出版社
北　京

图书在版编目（CIP）数据

Python高级机器学习 / （英）约翰•哈蒂（John Hearty）著 ; 王大鹏，管晨，郭鹏程译. -- 北京：人民邮电出版社，2020.1
（图灵程序设计丛书）
ISBN 978-7-115-52968-8

Ⅰ. ①P… Ⅱ. ①约… ②王… ③管… ④郭… Ⅲ. ①软件工具—程序设计 Ⅳ. ①TP311.561

中国版本图书馆CIP数据核字(2019)第290132号

内 容 提 要

Python 是杰出的脚本语言，在速度、安全性和可扩展性等方面表现出众。在机器学习领域，越来越多的从业者也选择使用易于上手、功能强大的 Python 语言。本书依次介绍无监督机器学习、深度信念网络、堆叠式降噪自编码机、卷积神经网络、半监督学习技术、文本特征工程，模型集成方法，以及其他一些实用的 Python 机器学习工具。除了理论知识与代码实现，本书还细致讲解了分析和解决问题的思路，极具启发性。

本书适合 Python 开发人员、分析师和数据科学家阅读。

◆ 著　　　　[英] 约翰•哈蒂
译　　　　王大鹏　管　晨　郭鹏程
责任编辑　岳新欣
责任印制　周昇亮
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷
◆ 开本：800×1000　1/16
印张：12
字数：283千字　　　　2020年 1 月第 1 版
印数：1 – 3 500册　　　　2020年 1 月北京第 1 次印刷
著作权合同登记号　图字：01-2017-9035号

定价：59.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

前　　言

你好！欢迎阅读这本基于 Python 的高级机器学习指南。你很可能是因为最初萌发的兴趣而翻开本书，但并不确定能从中获得什么。简单地说，学习和运用机器学习技术从未像现在这样激动人心，而且，在机器学习领域中工作会使你长期受益。要想了解更高级的数据建模技术的最新知识或积累应用这些技术解决难题的经验，本书便是你的绝佳选择！

什么是高级机器学习

受计算能力持续高速增长（根据摩尔定律）的影响，机器学习已经从曾经的研究型学科逐渐演化为活跃在各个商业领域的技术。这导致了新应用和新的（或重新发掘的）技术大量涌现，也使得原本不为大众所熟知的数据科学、人工智能和机器学习等概念迅速走进人们的视野，并成为全球企业战略规划的一部分。

机器学习应用的飞速发展得益于人们对持续创新的不断追求和大量研究资源的投入。这些先驱开发的技术开拓了新的应用领域，并让大众进一步了解了这些技术。人工智能和机器学习在个别领域的应用革新仍然很难实现，但有些则已经成为现实，如自动驾驶汽车、复杂图像的识别和修正、基因研究的更大胆尝试，而最为普遍的可能是在线商店、电子邮件收件箱和网络生活中逐渐成熟的个性化内容推荐。

面对众多可能性，在数据科学家专注的努力下，这个行业正经历着虽有些坎坷但飞速的成长。不但数据科学家和人工智能从业者的数量比几年前（2014 年初）多了很多，而且高端机器学习研究的解决方案也变得更开放和更容易获取。

谷歌和 Facebook 的研究团队逐渐开始共享其架构、语言、模型和工具，以期它们能得到更多数据科学家的应用和改进。

机器学习社区已经足够成熟，能够不断创造或重新挖掘许多先进算法，而且这种趋势已经日益明显。更准确地说，曾经主要属于学术界的研究方向如今已经得到了业界的高度重视。成果之一是，现在出现了很多跨越业界和学术界的机器学习专家。另一个成果是，人们逐渐认识到一些高级算法可以用于攻克当今的一些前沿难题。随着时间的推移，我们也目睹了不断取得的新进展，其效果越来越好，前沿边界也在逐步推进。

综上所述，如果你想进入数据科学领域并培养机器学习技能，那么现在就是最好的时机。有关入门算法（包括聚类、回归模型和神经网络架构）及其工具的内容可以在众多在线课程和各类博文中找到。虽然掌握最前沿的数据科学技术（包括深度学习、半监督算法和集成模型）依然有难度，但这些技术本身已经可以在多种编程语言的软件库中使用了。你需要做的只是学习如何将理论知识与实际指导相结合，从而正确应用各种模型，这正是本书被赋予的使命。

你将从本书中得到什么

本书致力于解读近年来出现的一些高级建模技术。无论你是经验丰富的数据科学家，还是想充分利用现有技能进入新领域的开发人员，都可以通过本书学习这些算法。

我的首要目标是确保你能够理解书中所探讨的算法。其中一些算法非常棘手，而且它们与统计学和机器学习中的其他概念紧密相关。

强烈建议新手先大致了解以下重要概念。

- 包括 MLP 架构在内的神经网络架构
- 包括梯度下降法和反向传播算法在内的学习算法
- 网络模型的性能评估指标，如均方根误差
- k 均值聚类

有时，本书对所涉及的主题探讨得不够充分。因为本书的内容范围较广，所以进度会略快。在每章末尾，我都会推荐一些书或在线文章作为补充阅读材料，以便你就相关知识建立更广泛的基础。在阅读本书的过程中，如果遇到任何陌生概念，建议你进行一些额外的阅读，因为机器学习的知识之间具有协同的关联性；掌握得越多，就越容易在积累新的知识与技能时理解新概念。

持续积累新的知识与技能这一理念对于本书要达成的目标至关重要。每章都会介绍一种或多种算法，以期达成以下目标：

- 从高层次上解释某算法的功能、适合解决的问题及其应用方法
- 浏览算法的关键部分，包括拓扑结构、学习方法和性能评估
- 通过评估模型输出确定如何提升性能

除了传授知识和技能，本书还希望达到一个更重要的目的：讨论和传达出资深机器学习从业者常见的一些品质。其中一种品质是创造力，主要体现在定义复杂结构和处理特定问题中。另一种重要品质则是严谨，本书通篇强调了这种品质，特别是在有意义的目标的性能测定和早期工作的严格评估方面。

最后，本书无意掩盖挑战数据难题时的现实：早期试验好坏参半的结果、大量的反复计算和频繁出现的绝境。与此同时，我会用一系列示例、对专业方法的剖析，以及更符合现实的难题

（直到本书末尾）来展示如何用具有创造性、可靠和严谨的方法破除这些障碍，从而取得有意义的结果。

希望你在阅读过程中一切顺利，也希望你能够在解决为你准备的问题、将所学知识应用于新领域或新数据的过程中获得乐趣。

我们现在就开始吧！

本书导读

第 1 章，无监督机器学习，介绍了如何用无监督学习技术来识别数据集的内在模式与结构。

第 2 章，深度信念网络，解释了受限玻尔兹曼机算法和深度信念网络算法的工作原理。通过学习这一章，你将了解如何应用这些算法，且更有把握通过评估输出结果来实现算法模型的改进与优化。

第 3 章，堆叠式降噪自编码机，通过用堆叠式降噪自编码机学习高维输入数据的特征表示，进一步提高你对深度架构的掌握程度。

第 4 章，卷积神经网络，展示了如何应用卷积神经网络（Convnet）。

第 5 章，半监督学习，阐释了如何应用多种半监督学习技术，如 CPLE、自主学习和 S3VM。

第 6 章，文本特征工程，探讨了一些数据预处理技巧，用于显著提高前述模型的性能和有效性。

第 7 章，特征工程 II，展示了如何清除或缓解数据质量问题来实现数据治理，以及如何针对具体的算法进行数据转换，并介绍了增强数据的一些技巧。

第 8 章，集成方法，探究了如何构建更复杂的集成模型，并介绍了提升模型结果稳健性的方法。

第 9 章，其他 Python 机器学习工具，回顾了近期数据科学家使用的最佳工具，阐述了各工具的优势，并以贯穿全书的工作过程讨论了如何配合前文介绍的工具和技术来应用这些工具。

附录，代码运行要求，讨论了本书所需要的工具，且分别指明了每一章所需要的库。

阅读前提

本书所使用的数据和代码都是公开的，包括开源 Python 库和框架。虽然每一章的示例代码都附有一个 README 文件，用于指明运行该章附带脚本中的代码需要安装的库，但我还是会将文件内容整理出来，以便你查阅。

建议你在阅读前几章时就安装好所需要的库，以便运行后续各章中的代码时使用。最好能提早安装第 1 章所需要的库，以便后文使用。

读者对象

本书面向渴望了解近期数据科学领域中最新的进展，以提高自身技能的 Python 开发人员、分析师和数据科学家。例如，如果你想构造自己的图片或文本标注解决方案，或者想要参加 Kaggle 大赛，那么本书就是为你而写！

如果具备一定的 Python 功底并了解机器学习的一些核心概念，那么你将可以更轻松地阅读本书。

排版约定

本书采用多种文本样式，以表示不同类型的信息。以下是一些样式的例子及其含义。

正文中的代码、数据库表名、用户输入将按如下样式显示：“通过使用下列代码，我们将 PCA 应用到手写 `digits` 数据集。”

代码块的格式如下所示：

```
import numpy as np
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
from sklearn.lda import LDA
import matplotlib.cm as cm

digits = load_digits()
data = digits.data

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target
```

命令行输入或输出的格式如下所示：

```
[ 0.39276606  0.49571292  0.43933243  0.53573558  0.42459285
  0.55686854  0.4573401   0.49876358  0.50281585  0.4689295 ]

0.4772857426
```

> 此图标表示警告或重要注释。

此图标表示提示或技巧。

读者反馈

欢迎你随时提供阅读反馈，告诉我们你对本书的看法。读者反馈对我们很重要，它可以帮助我们开发更多读者能够真正充分利用的图书。

如有常规问题的反馈，请发送邮件至 feedback@packtpub.com，并在主题处注明书名。

如果你擅长某个话题，而且愿意撰写相关文章或投稿，请登录 www.packtpub.com/authors 阅读我们的作者指南。

用户支持

现在你已经成为 Packt 出版社作品的拥有者，因此我们为你准备了一系列内容，以帮助你充分发挥所购图书的作用。

下载示例代码

如果你是从 http://www.packtpub.com 网站购买的图书，登录自己的账号后就可以下载所有已购图书的示例代码。如果你是从其他地方购买的图书，请访问 http://www.packtpub.com/support 网站并注册，我们会将代码文件直接发送到你的电子邮箱。

你也可以通过以下步骤下载代码文件。

(1) 使用你的电子邮箱地址和密码在我们的网址上登录或注册。

(2) 将鼠标移到页面上方的 SUPPORT 标签。

(3) 点击 Code Downloads & Errata。

(4) 在 Search 框中输入书名。

(5) 选择你需要下载代码文件的图书。

(6) 在下拉菜单中选择你购买本书的途径。

(7) 点击 Code Download。

下载文件后，确保使用以下工具的最新版本来解压或提取文件夹：

- WinRAR / 7-Zip（Windows）

- Zipeg / iZip / UnRarX（Mac）
- 7-Zip / PeaZip（Linux)

本书代码也托管在 GitHub 上，访问 https://github.com/bonzanini/Book-SocialMediaMiningPython 即可获取。Packt 拥有丰富的图书和视频资源，GitHub 仓库 https://github.com/PacktPublishing/提供了这些资源的相关代码。欢迎查阅！

下载本书的彩色图片

我们也为你提供了一份 PDF 文件，其中包含了本书中的截屏和图表等彩色图片。这些彩色图片可以帮助你更好地理解输出的变化。下载地址：https://www.packtpub.com/sites/default/files/downloads/AdvancedMachineLearningwithPython_ColorImages.pdf。

勘误

虽然我们竭力确保图书内容的正确性，但错误在所难免。如果你在我们出版的任何一本图书中发现了文本或代码错误，希望你能告知我们，我们将非常感激。你的善举足以减少其他读者在阅读出错内容时的纠结和不快，并帮助我们在后续版本中更正错误。如果你发现任何错误，请访问 http://www.packtpub.com/submit-errata，选择相应图书，点击 Errata Submission Form 链接，并输入有关错误的具体信息。[①]提交的错误得到验证后，我们就会接受你的建议，并将该处错误信息上传到我们的网站或添加到已有勘误表的相应位置。

访问 https://www.packtpub.com/books/content/support 并在搜索框中输入书名，即可查看该图书已有的勘误信息。这部分信息会在 Errata 部分显示。

反盗版

任何媒体都会面临版权内容在互联网上的盗版问题，Packt 也不例外。Packt 非常重视版权保护。如果你发现我们的作品在互联网上被非法复制，不管以什么形式，都请立即为我们提供相关网址或网站名称，以便我们进行补救。

请将疑似盗版材料的网址发送到 copyright@packtpub.com。

你的反盗版行动就是在保护作者和出版社，只有这样，我们才能继续以优质内容回馈像你这样的热心读者。

① 本书中文版勘误请到 https://www.ituring.com.cn/book/1941 查看和提交。——编者注

问题

如果对本书存有任何方面的疑问，可以通过 questions@packtpub.com 邮箱联系我们，我们将尽力为你答疑解惑。

致谢

我要感谢许多人，尤其是我的父母……感谢他们的耐心。还要感谢 Tyler Lowe 真挚的友情，感谢 Mark Huntley 对准确性的再三强调，感谢我之前在 Lionhead 工作室的团队。我还非常感谢 Jared Huffman 和 Packt 出版社勤勉的编辑团队所做的出色工作，在本书的创作过程中，他们一直给予我极大的信心和支持。

最后，我想把这本书献给你，我的读者。若想掌握本书的主题，没有比现在更好的时机了。如今的世界充满新机遇，而你可以用创造性的头脑和恰当的模型来抓住机会。希望你在寻求解决方案时，总是能够取得成功。

电子书

扫描如下二维码，即可购买本书电子版。

目　　录

第 1 章 无监督机器学习

本章将介绍如何用无监督学习技术来识别数据集中的模式和结构。

无监督学习是用于探索性分析的一系列很有价值的技术。它们能够挖掘出隐藏在数据集中的模式和结构，获取有用的信息或为进一步分析提供指导。拥有一套可靠的无监督学习工具集至关重要，你可以应用它们将陌生或复杂的数据集分解为可用信息。

我们将先从**主成分分析**（principal component analysis，PCA）这个基本的数据操作技术开始，并结合一系列的降维应用。接着探讨一项应用广泛且简单易用的无监督学习技术——***k*** **均值聚类**。随后会讨论一种能将复杂数据集映射到二维的拓扑聚类方法——Kohenen 提出的**自组织映射**（self-organizing map，SOM）。

本章会用一些篇幅来探讨如何有效应用这些技术将高维数据集整合为易于处理的形式。我们将使用 **UCI 手写数字**数据集来展示每种算法的技术应用。在探讨和应用每种技术的过程中，我们将回顾一些实际应用和方法论的问题，尤其是如何校准和验证每种技术，以及哪些性能评测方法是有效的。综上，我们将按顺序讨论以下主题：

- 主成分分析
- *k* 均值聚类
- 自组织映射

1.1 主成分分析

为了有效处理高维数据集，我们必须有一套技术能将维度降至可处理的水平。降维的优势包括：可以在二维空间中对多维数据进行可视化、用最少的特征描述数据集所包含的信息，以及在某些情况下识别模型向量的共线性。

> 在机器学习的背景下，所谓的共线性是指模型特征共有的接近线性的关系。这些特征一般用处不大，原因很明显，相关特征不大可能比它们独立时提供更多的信息量。而且，共线特征可能会突出局部极小值或给出其他错误的线索。

当今应用最广泛的降维技术可能就是主成分分析了。本书将在多处使用主成分分析，因此最好先回顾一下该技术、理解其背后的理论，并编写能够有效应用主成分分析的 Python 代码。

1.1.1　主成分分析入门

主成分分析是一种强大的分解方法，它能够将变量极多的数据集分解为一系列正交向量。取足够多的正交向量，就能够解释数据集中近乎全部的方差了。从本质上来说，集合中的这些正交向量就是数据集的精简版描述。主成分分析用途广泛，很值得我们花时间学习。

> 这里稍微提醒一下，当分解后的向量维度小于原始数据集的变量个数时，往往会损失一些原始数据集中的信息。若向量数目足够多，这种损失通常很小，但如果将变量极多的数据集分解成很少的向量，那么就会损失“惨重”。因此，在应用主成分分析时，必须考虑对数据集有效建模所需要的向量数目。

主成分分析通过逐一辨别数据集中方差最大的方向（主成分）来提取向量，步骤如下所示：

(1) 找出数据集的中心点；
(2) 计算数据的协方差矩阵；
(3) 计算协方差矩阵的特征向量；
(4) 将特征向量标准正交化；
(5) 计算每个特征向量表示的方差比例。

我们来简要介绍一下这些概念。

- **协方差**：多维数据的方差，两个或多个变量之间的方差。一维或一个变量的方差可由一个值来描述，两个变量之间的方差需要用一个 2×2 的矩阵来描述，三个变量之间的方差需要用一个 3×3 的矩阵来描述，以此类推。因此，主成分分析的第一步就是计算协方差矩阵。
- **特征向量**：一种针对数据集或线性变换的向量。具体来说，这种向量的方向在转换前后不会发生变化。为便于理解，假设你的两手之间有一根直的橡皮筋，你伸展橡皮筋，直到它绷紧为止。特征向量正是伸展前和伸展过程中不会发生方向变化的向量，也就是本例中直接穿过橡皮筋中心从一只手到另一只手的向量。
- **正交化**：寻找两个相互正交（成直角）的向量的过程。在 n 维数据空间中，正交化的过程就是选取一系列向量，然后转化为一系列正交向量。
- **标准正交化**：对积进行标准化的正交化过程。
- **特征值**：（大致相当于特征向量的长度）用于计算由每个特征向量表示的方差比例，该比例由每个特征向量的特征值除以所有特征向量的特征值之和得到。

总的来说，协方差矩阵用于计算特征向量，标准正交过程则是根据特征向量得出正交的、标准化向量。特征值最大的特征向量是第一个主成分，其他成分的特征值都相对较小。这样一来，主成分分析算法就能够将数据集转化为用新的低维坐标系来表示。

1.1.2 应用主成分分析

概略地回顾了主成分分析算法后，接下来我们将其应用于一个重要的 Python 数据集，即 UCI 手写数字数据集（`digits`），它是作为 scikit-learn 的一部分发布的。

该数据集包含了来自 44 位作者的 1797 个手写数字。对这些作者的手写数字的笔压和位置信息使用 8×8 的点阵，经过两次重复采样后，得到了图 1-1 中的映射。

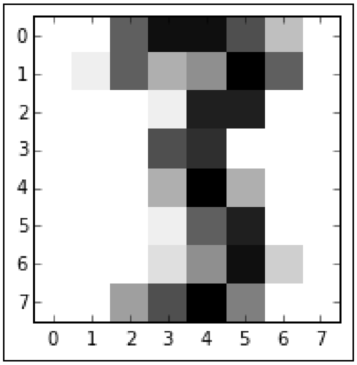


图 1-1

这些映射可以转化为维度为 64 的特征向量，从而作为分析输入项。对于含有 64 个特征的输入数据集，我们往往会立即想到使用诸如主成分分析的技术，从而将变量减少到可处理的数量。目前来看，我们无法用可视化探索来有效地分析数据集。

先对 `digits` 数据集应用主成分分析技术，代码如下所示：

```
import numpy as np
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale
from sklearn.lda import LDA
import matplotlib.cm as cm

digits = load_digits()
data = digits.data

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target
```

这段代码完成了以下工作。

首先，它加载了一系列必要的库，包括 numpy 库、scikit-learn 库和用于绘图的 matplotlib 库。其中 scikit-learn 库主要包括 `digits` 数据集、主成分分析算法和数据标准化函数等一系列相关数据、算法和函数。

接着，开始准备 `digits` 数据集。步骤如下所示。

- 首先，在构造有用的变量前加载数据集。
- 创建 `data` 变量用于表示数据集，并根据 `digits` 数据集中的目标向量计算出因变量的类别数（因为数字取值范围为 0~9，所以 `n_digits = 10`）并存为变量，以供后续分析使用。
- 将 `target` 向量存为变量，并命名为 labels，以便后续使用。
- 以上变量构造均是为了简化后续分析。

准备好数据集后，对主成分分析算法进行初始化并将其应用于数据集。

```
pca = PCA(n_components=10)
data_r = pca.fit(data).transform(data)

print('explained variance ratio (first two components): %s' %
                                str(pca.explained_variance_ratio_))
print('sum of explained variance (first two components): %s' %
                                 str(sum(pca.explained_variance_ratio_)))
```

以上代码会输出由前 10 个主成分解释的方差，按解释能力由高到低排序。

本例中的 10 个主成分共同解释了 0.589 的总方差。这结果不算太差，毕竟我们将 64 个变量减少到 10 个主成分，但也的的确确反映出了主成分分析带来的潜在损失。然而，关键问题是，精简后的变量集能否便于后续的分析或分类，也就是说，降维向量保留的方差信息是否会影响分类预测效果。

用 `pca` 处理 `digits` 数据集，并将输出结果保存到新变量 `data_r` 中后，可以对输出进行可视化。为此，首先需要构建 `colors` 向量来给类上色，然后便可以绘制散点图：

```
X = np.arange(10)
ys = [i+x+(i*x)**2 for i in range(10)]

plt.figure()
colors = cm.rainbow(np.linspace(0, 1, len(ys)))
for c, i target_name in zip(colors, [1,2,3,4,5,6,7,8,9,10], labels):
    plt.scatter(data_r[labels == I, 0], data_r[labels == I, 1],
                c=c, alpha = 0.4)
    plt.legend()
    plt.title('Scatterplot of Points plotted in first \n'
    '10 Principal Components')
    plt.show()
```

得到的散点图如图 1-2 所示。

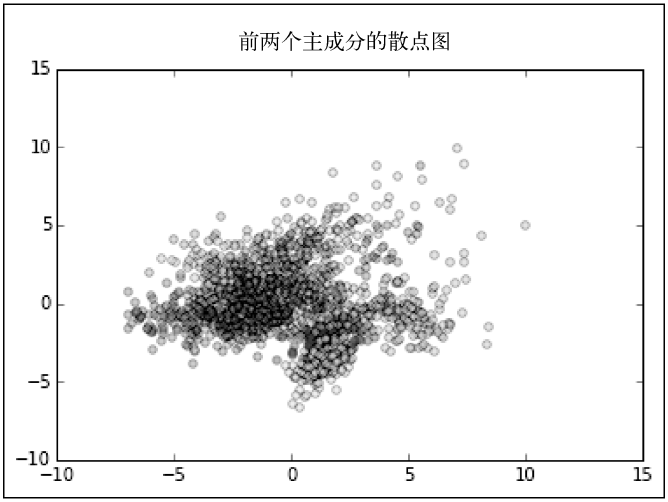


图 1-2

这幅图告诉我们，虽然前两个主成分之间有一些间隔，但很难用本数据集来高度精确地区分。然而，很多类的内部的确出现了一些聚合现象，我们可以通过聚类分析获取一些有用信息。照此方法，我们可以利用主成分分析初步了解数据集的结构，然后再进一步分析。

接下来，应用 k 均值聚类算法来继续探索聚类结果。

1.2 *k* 均值聚类

前面已经介绍过，无监督机器学习算法通常用于从复杂的大型数据集中提取关键的结构或内容信息。这些算法对手动输入的要求很低或无要求，并且不需要训练数据（即用一系列带标签的解释变量和响应变量来训练算法，以识别所需要的分类边界）。这意味着，无监督算法能有效生成新数据集或陌生数据集的结构和内容信息，以便分析师在短时间内深入理解数据。

1.2.1 聚类入门

聚类可能是最典型的无监督学习技术，原因如下。

聚类算法经过了长久而深入的发展和优化，包括 Python 在内的大部分数据科学语言都进行了高效实现。

聚类算法执行高效，算法实现的平滑复杂度在多项式级别。这使得同时执行多个聚类过程变得没有那么复杂，即便是在大规模数据集上。另外，可扩展的实现方法也是存在的，它们可以在

TB 级规模的数据集上并行运行聚类算法。

聚类算法通常比较容易理解，若有必要，其操作过程也比较容易解释。

最常用的聚类算法是 k 均值。通过在数据空间中随机确定 k 个点，该算法可以构建 k 个簇，每个点都是对应簇的均值。然后可以进行如下循环过程：

- 根据（簇内）最小平方和原则，即距离最近原则，将数据点分配至不同簇；
- 将每个簇的中点（质心）作为新的均值，这会导致均值的位置发生变化。

经过足够多的迭代后，质心就会移到能使性能指标（最常用的是“簇内最小平方和”指标）最小化的位置，此时观测点不会再随迭代而变化，算法也得到了收敛，从而得出聚类的一个解。

1.2.2　开始聚类分析

回顾过聚类算法后，接下来运行以下代码并观察聚类的效果。

```
from time import time
import numpy as np
import matplotlib.pyplot as plt

np.random.seed()


digits = load_digits()


data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

sample_size = 300

print("n_digits: %d, \t n_samples %d, \t n_features %d" % (n_digits, n_samples,
n_features))


print(79 * '_')
print('% 9s' % 'init''          time   inertia    homo   compl   v-meas
ARI     AMI silhouette')

def bench_k_means(estimator, name, data):
    t0 = time()
    estimator.fit(data)
    print('% 9s %.2fs %i %.3f %.3f %.3f %.3f %.3f %.3f'
       % (name, (time() - t0), estimator.inertia_,
          metrics.homogeneity_score(labels, estimator.labels_),
          metrics.completeness_score(labels, estimator.labels_),
          metrics.v_measure_score(labels, estimator.labels_),
```

```
metrics.adjusted_rand_score(labels, estimator.labels_),
metrics.silhouette_score(data, estimator.labels_,
   metric='euclidean',
   sample_size=sample_size)))
```

> 上面的代码和之前的主成分分析代码的关键区别是，这里的代码先对 digits 数据集应用了标准化函数，将数据集的值限制在 0~1。必须在必要时对数据进行标准化，既可以是对数标准化，也可以是边界限制的标准化，以防不同量级的变量值对数据集造成不成比例的重大影响。数据是否需要进行标准化（以及何种标准化、标准化到什么范围等），很大程度上取决于数据的形态和性质。如果数据的分布中出现了离群点或大范围的波动，那么应用对数标准化更为合适。通过可视化和探索性分析还是统计分析结果来进行手动标准化，标准化决策往往与数据以及使用的分析方法紧密相关。如果想要进一步了解标准化的决策和考虑，参见第 7 章。

scikit-learn 的优点是默认应用 *k* 均值++算法。*k* 均值++算法在原始 *k* 均值算法的基础上进行了优化，兼顾了运行时间效率和聚类成功率，以避开糟糕的聚类结果。

k 均值++算法通过运行初始化过程来寻找逼近簇内最小方差的簇质心。

你可能已经从前面的代码中看出，我们用了一系列性能评估指标来跟踪 *k* 均值聚类的效果。仅依赖正确率或其他算法常用的类似性能指标来评估聚类算法是不现实的。聚类算法的成功可以定义为它能够解释数据中蕴含的分组方式，而这些方式会受到包括类的分离度、组间相似性和组间差异在内的各种因素的影响。

同质性得分是一个取值在 0~1 的简单指标，用于度量簇内的数据点仅归属于某一类的程度。1 代表所有簇内的数据点都属于同一类。该指标可与**完整性得分**一起使用，后者是一个类似的有界指标，度量给定类的成员被分配到同一个簇的程度。因此，完整性得分和同质性得分都为 1 的聚类就是完美的聚类。

有效性测度（v 值）是同质性得分和完整性得分的调和平均数，与二元分类的 F 值非常类似，其本质上是用一个 0~1 范围内的值来代表聚类的同质性和完整性。

调整兰德指数（Adjusted Rand Index，ARI）是一个相似性指标，用于表示数据点集合间的一致性。对聚类来说，该指标度量的是已知的真实数据标签和基于聚类算法的预测标签之间的一致性。兰德指数在 0~1 有界范围内度量标签相似性，其中 1 代表完美地预测了标签。

上文提及的所有性能指标和其他相似指标（如赤池互信息准则）的主要问题是，它们都要基于对真实状况的理解，也就是说，被分析的部分数据或所有数据需要具有标签。如果标签不存在或无法生成，则这些方法无效。在实际工作中，这是一个极大的劣势，因为极少数据集是事先标

注的，而且标注数据的过程非常耗时。

无须标注数据就能度量 k 均值聚类效果的一种指标是**轮廓系数**。它能度量模型内聚类的明确程度。数据集的轮廓系数指的是每个样本系数的均值，其中系数的计算方法如下所示：

$$s = \frac{b - a}{\max(a, b)}$$

公式中每项的定义如下所示。

- ❑ a：样本和簇内其他所有数据点的距离的均值。
- ❑ b：样本和最近邻簇内所有数据点的距离的均值。

该式取值范围为–1~1，其中–1 代表错误聚类，1 代表密度极大的聚类，0 代表重叠聚类。上述属性与我们对一个良好聚类方案的期望相吻合。

可以对 `digits` 数据集应用前面提及的所有性能测量方法。因此，可以在 `digits` 数据集上初始化 `bench_k_means` 函数来完成前面的示例：

```
bench_k_means(KMeans(init='k-means++', n_clusters=n_digits, n_init=10),
              name="k-means++", data=data)
print(79 * '_')
```

这段代码会得到如图 1-3 所示的输出（注意，随机数种子的不同会导致结果不同）。

```
n_digits: 10,    n_samples 1797,        n_features 64
_______________________________________________________________________________
init         time   inertia    homo   compl  v-meas     ARI     AMI  silhouette
k-means++   0.25s    69517   0.596   0.643   0.619   0.465   0.592    0.123
_______________________________________________________________________________
```

图　1-3

我们来仔细地观察一下这个结果。

轮廓评分为 `0.123`，相当低，但在意料之中，毕竟手写数字数据本来就有很多噪声，的确容易出现重叠。然而，其他一些分数就不那么突出了。`0.619` 的 v 值是比较合理的，但同质性得分很糟糕，这意味着簇质心不够完美。另外，`0.465` 的 ARI 也不够理想。

> 这么说吧，最差的分类（即随机分类）最高能达到 10%的分类精度。所有性能测量指标的值也会非常低。虽然肯定比随机分类更准确，但跟最优的分类算法相比，我们所达到的效果还差得远。第 4 章会介绍，当卷积神经网络应用于手写数字数据集时，分类错误将非常少。传统的 k 均值聚类几乎无法达到这样的精度水平。

总而言之，有理由相信，我们可以做得更好。

因此我们再试一次，这一次多加一个处理环节。先应用前面用过的主成分分析技术为输入数据集降维。代码很简单，如下所示：

```
pca = PCA(n_components=n_digits).fit(data)
bench_k_means(KMeans(init=pca.components_, n_clusters=10),
              name="PCA-based",
              data=data)
```

这段代码将 PCA 应用到 digits 数据集，所获得的主成分数与数据集的标签类别数（本例即为数字的种类数）一致。进行下一步前，我们应该先检查一下 PCA 的输出结果，因为主成分维度过低可能代表数据集中存在共线性，或者需要进一步研究。

这一次的聚类效果得到了显著提升（见图 1-4）。

n_digits: 10, n_samples 1797, n_features 64

init	time	inertia	homo	compl	v-meas	ARI	silhouette
PCA-based	0.02s	71820	0.673	0.715	0.693	0.567	0.121

图 1-4

v 值和 ARI 大约提高了 0.08，v 值甚至达到了相当优秀的 0.693。轮廓系数并没有发生很大改变。因为数据集非常复杂，而且簇间覆盖非常严重，所以现有结果已经很理想了，尤其是只添加一小段简单代码就得到了这种提升！

对 digits 数据集进行聚类后，就可以看到一些有意义的簇。图 1-5 同样表明，很难从输入的特征向量中挖掘出特征。

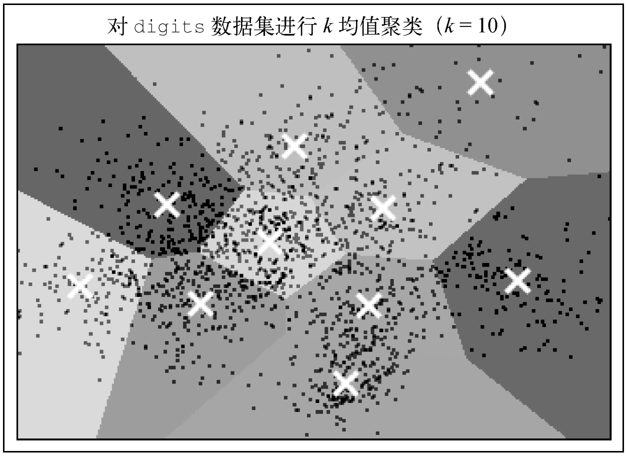


图 1-5

1.2.3 调整聚类参数

前面的示例描述了 k 均值聚类的应用，并用相关代码展示了如何将聚类分析结果可视化，还介绍了一些适用的性能度量指标。然而，对真实的数据集应用 k 均值聚类时，还必须考虑其他问题，后文会对此进行讨论。

另一个关键点是，如何选择恰当的 k 值。以特定 k 值对 k 均值聚类进行初始化没有坏处，但很多情况下，我们一开始可能并不知道最终要确定多少个簇，因而不知道将 k 定为多少合适。

可以用不同的 k 值重新运行前面的代码，并观察性能测量方法的值，但这无法告诉我们最能捕捉数据特征的 k 值是多少。k 值调整存在的风险是：随着 k 的增大，轮廓系数或未解释方差可能会大幅减小，但我们无法得到有意义的聚类。极端情况是，当 $k=o$ 时（o 为样本数据的观测数），即每个数据点都自成一簇时，轮廓系数会很小，但聚类结果毫无意义。同时，还有许多 k 值过高导致过度拟合的例子。

为缓解这种风险，可以用几种辅助方法（如**肘部方法**）先确定 k 的几个可能取值。肘部方法很简单，绘制解释方差随 k 值变化的图像即可。这样的图像一般看起来很像一只弯曲的手臂。

对主成分分析降维过的数据集来说，代码如下所示：

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from scipy.spatial.distance import cdist
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

K = range(1,20)
explainedvariance= []
for k in K:
    reduced_data = PCA(n_components=2).fit_transform(data)
    kmeans = KMeans(init = 'k-means++', n_clusters = k, n_init = k)
    kmeans.fit(reduced_data)
    explainedvariance.append(sum(np.min(cdist(
    reduced_data, kmeans.cluster_centers_, 'euclidean'), axis = 1))/data.shape[0])
    plt.plot(K, meandistortions, 'bx-')
    plt.show()
```

以上代码应用了肘部方法，在前面 `PCA` 降维的相关示例代码的基础上检验了解释方差（更具体地说，检验了簇内解释方差）。输出结果是给定取值范围内的所有 *k* 值及其对应的未解释方差的度量。在本例中，因为使用的是 `digits` 数据集（已知包含 10 个类），所以 *k* 的取值范围为 `1~20`（见图 1-6）。

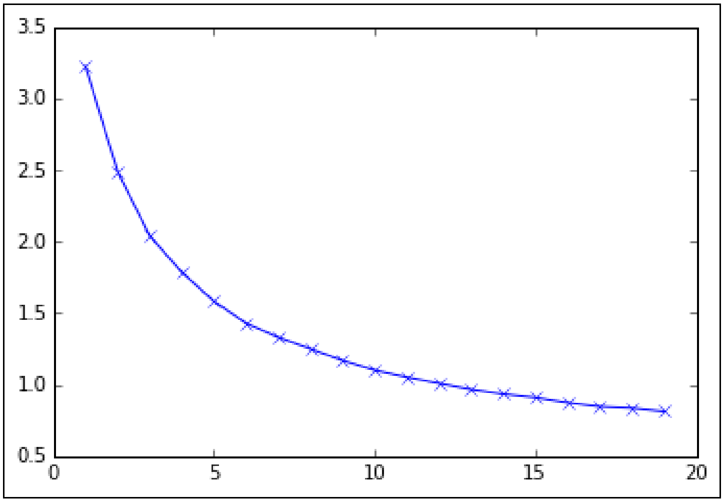


图　1-6

肘部方法的原理是，选择能使解释方差达到最大、*k* 达到最小的值。其中 *k* 是“肘”的拐弯处的值。肘部方法的技术性含义是，随着 *k* 的增大，解释方差的最小增长会与过度拟合的风险相抵消。

但肘部方法图的效果有时可能不够显著，因为无法确保“肘部”永远清晰易辨。本例中的曲线比较平滑，根据其他数据集绘制的肘部方法图可能更易于观察。值得注意的是，虽然我们知道数据集中类的数目是 10，但肘部方法表明，从一开始就出现了 *k* 值增大所带来的收益递减现象，且“肘部”大约位于类个数为 5 的地方。这与类之间存在大量重叠有关，前面的图像中已经体现了这一点。虽然总共有 10 个类，但要清楚识别 5 个以上的类将会变得越来越困难。

还需要注意的是，肘部方法本质上是一种启发式探索，而非什么客观原则。作为一种提升聚类效果的预处理方法，主成分分析也会将图像变平滑，从而得到更为平缓、循序渐进的曲线。

除了应用肘部方法，还可以观察聚类本身（像本章前面那样做），用主成分分析对数据降维。对数据集进行可视化、对数据映射聚类分配后，往往很容易发现，*k* 均值聚类何时能找到局部最小值、何时会过度拟合。图 1-7 展示了对 `digits` 数据集应用前文中的 *k* 均值聚类算法时出现的过度拟合现象，此时规定 *k*=150。在本例中，有的簇只包含 1 个观测值，因此该簇完全无法推广到其他样本。

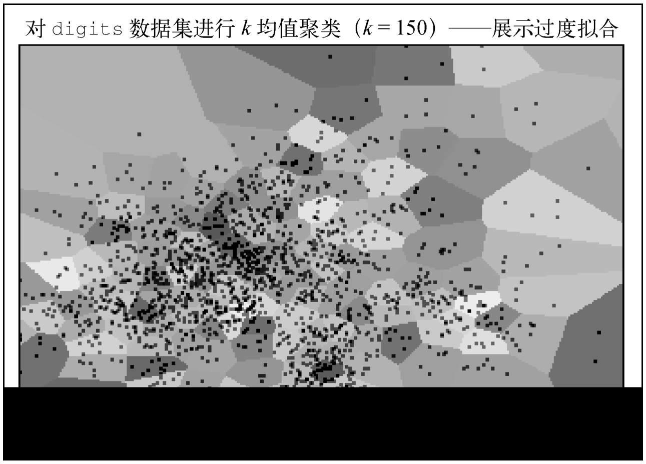


图　1-7

绘制肘部曲线或聚类分配比较快速易行，而且易于理解。但前文提过，这些技术只是一种启发式探索。如果数据集中的类数目明确，那么就无法确保这种启发式探索能够带来可归纳的结果。

这种方法的另一个缺点是，通过绘图检验聚类效果非常手工化，并不适合生产环境或自动化。在这些情况下，更理想的方法是基于代码的可自动化方法，比如应用广泛的 **v 重交叉验证**。

交叉验证非常容易上手。首先，将数据集分为 v 份，其中一份用作测试集，然后用剩下的 $v-1$ 份数据来训练模型。接下来我们继续用 digits 数据集试试手：

```
import numpy as np
from sklearn import cross_validation
from sklearn.cluster import KMeans
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale


digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target



kmeans = KMeans(init='k-means++', n_clusters=n_digits, n_init=n_digits)
cv = cross_validation.ShuffleSplit(n_samples, n_iter = 10, test_size = 0.4,
                                   random_state = 0)
scores = cross_validation.cross_val_score(kmeans, data, labels, cv = cv,
                                          scoring = 'adjusted_rand_score')
print(scores)
print(sum(scores)/cv.n_iter)
```

以上代码执行了一些你已经很熟悉的数据加载和数据准备过程，并且初始化了 *k* 均值聚类算法。然后它定义了 `cv`，即交叉验证的参数。这包含了循环次数 `n_iter`，以及每次交叉验证的测试数据占比。在本例中，我们将用 60%的数据样本进行训练，剩下的 40%用于测试。

然后应用 *k* 均值模型，将已定义好的 `cv` 参数代入交叉验证评分函数，并定义输出结果为 `scores`。如下所示：

```
[ 0.39276606  0.49571292  0.43933243  0.53573558  0.42459285
  0.55686854  0.4573401   0.49876358  0.50281585  0.4689295 ]

0.4772857426
```

输出项依次包含 *k* 均值++聚类 10 重交叉验证中每次验证得到的调整兰德评分。该评分在 0.4~0.55 范围内波动，应用主成分分析前的 *k* 均值++聚类的 ARI 也落在这个区间中（为 0.465）。以上代码可以融入我们的分析，以实现在底层持续变化的情况下自动评估聚类性能。

本章前面提过，测量聚类效果的方法应与所掌握的信息一致。大多数情况下，我们无法得到数据集的实际标签，因此不得不使用诸如轮廓系数的方法。

> 有时，即使使用交叉验证和可视化，也无法得出明确的结论。尤其是在处理陌生数据集时，有时会碰到这种问题：对于不同的 *k* 值，算法可能会错误解析某些噪声或者次级信号，并掩盖需要分析的信号。
>
> 与本书中讨论的其他所有算法一样，必须先了解需要处理的数据集，否则再严谨、技术再正确的分析，都有可能得出错误的结论。第 6 章将进一步探讨陌生数据集的数据检查和数据准备的原则和技术。

1.3　自组织映射

自组织映射是一种基于拓扑表示法的数据降维技术。这样的技术有很多，其中主成分分析便是比较有名的一种。然而，作为降维技术和可视化形式，自组织映射自有其独到之处。

1.3.1　自组织映射入门

自组织映射算法需要反复迭代一些简单的操作。在目标数据集很小时，自组织映射与 *k* 均值聚类非常类似（后面很快会讲到）；目标数据集较大时，自组织映射就会用一种强大的方式展现复杂数据集的拓扑结构。

自组织映射是由许多节点构成的网格（一般是矩形或六边形），每个节点都包含一个权重向量，该权重向量的维度与输入数据集相同。节点可以随机初始化，但与数据集分布大致相似的初始化能令算法训练得更快。

算法会随观测值的输入而迭代，迭代的形式如下所示。

- 找出现有配置下的激活节点，即**最佳匹配单元**（best matching unit，BMU）。通过度量输入向量与所有权重向量间的欧几里得距离，可以决定BMU[①]。
- 根据输入向量调整 BMU（将 BMU 向输入向量移动）。
- 通常以较小量调整临近节点[②]，临近节点调整的幅度由邻域函数决定。（邻域函数有很多种，本章将使用高斯邻域函数。）

该过程可能会重复多次，并运用合适的取样方法，直到网络收敛（当新输入项增加时，损失无法进一步降低）。

与神经网络的节点不同，自组织映射的节点通常会包含一个长度与输入数据集维度相同的权重向量，也就是说，输入数据集的拓扑结构可以用一个低维映射来存储并进行可视化。

自组织映射类实现的相关代码参见本书仓库中的 som.py 脚本。目前，我们先在熟悉的环境下应用自组织映射算法。

1.3.2　部署自组织映射

前面提到过，自组织映射算法是迭代的，建立在向量间欧几里得距离之上。

该映射会形成一个清晰易读的 2D 网格。就通用的鸢尾花教学数据集来说，自组织映射可以非常简洁地进行映射（见图 1-8）。

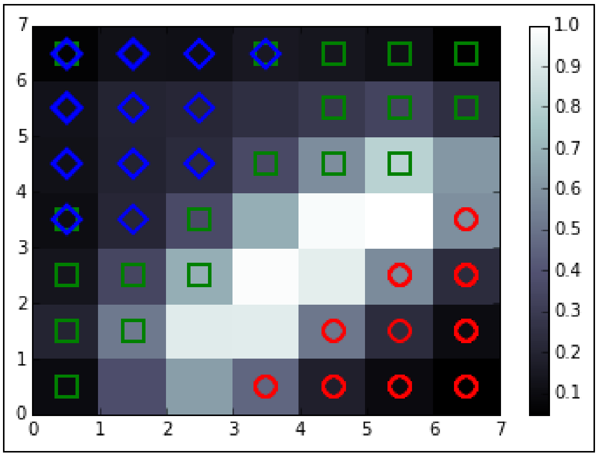


图　1-8

① 与输入向量距离最近的权重向量所在节点即最佳匹配单元。——译者注
② 即优胜邻域内的节点，其中优胜邻域也是由邻域函数决定的。——译者注

图中的类得到了有效划分，且按空间进行了排序。此例中的背景色代表的是聚类密度。蓝色类和绿色类出现了细微的重叠，此处自组织映射的分类也不够理想。在鸢尾花数据集中，自组织映射会在 100 次迭代后得到收敛的解决方案，而迭代增加到 1000 次时，效果提升并不明显。对分类不清的更复杂的数据集来说，该过程可能会持续成千上万次迭代。

尴尬的是，Python 目前没有类似 scikit-learn 这种包含自组织映射算法的包。因此，我们必须自己动手实现。

我们将处理的自组织映射代码位于相应的 GitHub 库中。现在我们先来看一下相关脚本，并大致理解代码的运行。

```
import numpy as np
from sklearn.datasets import load_digits
from som import Som
from pylab import plot,axis,show,pcolor,colorbar,bone


digits = load_digits()
data = digits.data
labels = digits.target
```

至此，我们就加载好了 `digits` 数据集，并根据数据标签定义了变量 `labels`。这样做可以方便我们观察自组织映射算法是如何通过低维映射进行类区分的。

```
som = Som(16,16,64,sigma=1.0,learning_rate=0.5)
som.random_weights_init(data)
print("Initiating SOM.")
som.train_random(data,10000)
print("\n. SOM Processing Complete")

bone()
pcolor(som.distance_map().T)
colorbar()
```

这里我们使用了 Som.py 文件中定义的 `Som` 类，该类包含了我们用来实现自组织映射算法的方法。我们将提供映射的维度（经过一系列尝试后，本例将 16×16 作为网格规模，因为这样的网格规模能够给特征映射留有充足的延伸空间，即使仍可能存在一些簇间的重叠）和输入数据的维度（用来确定自组织映射节点的权重向量的维度）作为函数的参数，同时提供 sigma 值和学习率。

本例中的 sigma 定义了邻域函数的覆盖范围。如前文所述，我们使用的是高斯邻域函数。最恰当的 sigma 值与网格规模有关。对 8×8 的网格来说，sigma 的值一般取 1.0，而本例 16×16 的网格对应的最佳 sigma 值为 1.3。sigma 取值不恰当是很容易发现的。当 sigma 过小时，数据点会在网格的中心聚集；当 sigma 过大时，网格中心就会出现若干较大空白。

不言自明，指标**学习率**指的是自组织映射的初始学习率。随着映射的迭代，学习率会根据以下公式进行调整：

$$学习率(t)=学习率/(1+0.5*t)$$

其中 t 为迭代指数。

先用随机权重来初始化我们的自组织映射。

> 与 k 均值聚类相同，自组织映射的随机初始化比基于近似数据分布的初始化要慢。应用与 k 均值++算法相似的预处理步骤，可以节约自组织映射的运行时间。目前来看，我们的自组织映射已经在 digits 数据集上运行得很快了，因此没必要进行这样的优化。

接下来，给每个类定义标签并分配颜色，以便在自组织映射图像中区分它们。然后遍历每个数据点。

每次迭代时，都要根据自组织映射算法的计算结果标出表明类的 BMU。

自组织映射结束迭代时，需要添加 **U 矩阵**（一种表示相对观测密度的颜色矩阵）作为一个单色图层：

```
labels[labels == '0'] = 0
labels[labels == '1'] = 1
labels[labels == '2'] = 2
labels[labels == '3'] = 3
labels[labels == '4'] = 4
labels[labels == '5'] = 5
labels[labels == '6'] = 6
labels[labels == '7'] = 7
labels[labels == '8'] = 8
labels[labels == '9'] = 9

markers = ['o', 'v', '1', '3', '8', 's', 'p', 'x', 'D', '*']
colors = ["r", "g", "b", "y", "c", (0,0.1,0.8), (1,0.5,0), (1,1,0.3),
          "m", (0.4,0.6,0)]
for cnt,xx in enumerate(data):
    w = som.winner(xx)
    plot(w[0]+.5,w[1]+.5,markers[labels[cnt]],
         markerfacecolor='None', markeredgecolor=colors[labels[cnt]],
         markersize=12, markeredgewidth=2)
    axis([0,som.weights.shape[0],0,som.weights.shape[1]])
    show()
```

以上代码可以得到的图像如图 1-9 所示。

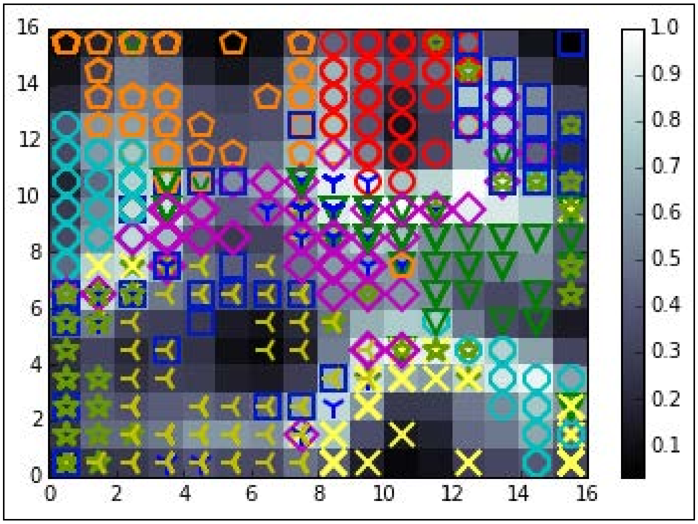


图 1-9

这段代码生成了一幅 16×16 个节点的自组织映射图像。由图可知，映射成功将每个簇分离成了拓扑上截然不同的图像区域。虽然其中几个类（尤其是蓝绿色圆圈代表的数字 5 和绿色星号代表的数字 9）出现在了自组织映射空间的许多位置，但大多数情况下，每个类都单独占据一块区域，这说明自组织映射还是非常有效的。U 矩阵表明，数据点密度较高的区域往往会包含多个类的数据，这一点我们在 *k* 均值聚类和主成分分析图像中早已见过，并不足为奇。

1.4 扩展阅读

Victor Powell 和 Lewis Lehe 提供了一份生动深刻、图文并茂的主成分分析讲解（http://setosa.io/ev/principal-component-analysis/），这份讲解很适合初次接触主成分分析核心概念的读者或还不太理解该概念的读者。

若想更深入地了解主成分分析及其背后的数学理论基础，并触及底层的矩阵变换原理，不妨阅读 Google research 的 Jonathon Shlens 提供的更清晰详尽的解释（http://arxiv.org/abs/1404.1100）。

若想将 Jonathon 版本的原理说明转化为 Python 代码以进行实践，Sebastian Raschka 以鸢尾花数据集所做的示范（http://sebastianraschka.com/Articles/2015_pca_in_3_steps.html）是一个不错的选择。

最后，如果想了解 sklearn 文档中与主成分分析类相关的参数细节，则可以参考 http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html。

David Robinson 有一篇有趣的博文“解释方差”（http://varianceexplained.org/r/kmeans-free-

lunch/）讲述了一个专业而生动的 k 均值聚类的实例，其中详细探索了导致聚类失败的条件，以及这种情况下可能的其他选择。

你还可以阅读 Rick Gove 对肘部方法的详细讨论（https://bl.ocks.org/rpgove/0060ff3b656618e9136b）。

最后，这份 sklearn 文档（http://scikit-learn.org/stable/tutorial/statistical_inference/unsupervised_learning.html）从另一个视角分析了包含 k 均值聚类在内的无监督学习算法，非常值得一读。

Kohonen 的自组织映射的许多现有阅读材料不是比较过时，就是过于高端，又或过于正式。你也可以替换阅读 John Bullinaria 的文章（http://www.cs.bham.ac.uk/~jxb/NN/l16.pdf）。

如果想更深入地理解背后的数学原理，我建议你直接阅读 Tuevo Kohonen 的文章。自组织映射 2012 年版就是很好的入门。

关于本章提到的多重共线性概念，https://onlinecourses.science.psu.edu/stat501/lesson/12/12.1 中的文章给出了清晰的解释。

1.5　小结

本章回顾了三种技术，它们在数据预处理和数据降维方面都有广泛的应用。在应用这些技术的过程中，就能够初步了解陌生数据集。

我们先回顾了主成分分析，这是一种应用广泛的降维技术，能够帮助我们理解并可视化高维数据集。接着，我们使用 k 均值聚类对数据进行了聚类，通过性能指标、肘部方法和交叉验证来识别改进和测量 k 均值分析的方法。当用同样的方法将 k 均值应用于 `digits` 数据集时，并没有得到什么特别的结果。通过主成分分析可视化，我们了解到这是因为数据集的类之间存在重叠。于是，我们使用主成分分析来进行数据预处理，以提升后续的聚类结果，从而克服了这一弱点。

最后，我们开发了一套自组织映射算法。对比主成分分析来说，这套算法可以对 `digit` 类做出更清晰的分割。

回顾完无监督学习技术和分析方法的关键基础知识后，接下来我们将探讨更加强大的无监督学习算法的用法。

第2章 深度信念网络

上一章介绍了一些广泛应用的降维技术，这些技术可以帮助数据科学家更深入地了解数据集的性质。

接下来几章将重点探讨深度学习领域中一些更复杂的技术。本章的目的是让你理解**受限玻尔兹曼机**（restricted Boltzmann machine，RBM）的应用，并掌握多个受限玻尔兹曼机链接生成的深度学习架构——**深度信念网络**（deep belief network，DBN）。经过训练，深度信念网络可以高效地解决文本、图像和声音识别中的复杂问题，通常被行业领先企业用于目标识别、智能图像搜索和机器人空间识别。

我们首先要做的是了解深度信念网络的底层算法，以打下坚实的知识基础。不同于聚类算法和主成分分析，深度信念网络代码在数据科学界的知名度并不高，因此我们先来回顾一下该算法，以确保你能够牢固掌握部分相关知识。了解理论之后，我们会在此基础上开始学习代码，巩固理论知识，并将技术应用到实际中。对这些技术的诊断非常重要，而且要求非常严谨，因此我们会强调一些思维过程和诊断方法，以便高效观察和管理这些技术的实现。

学完本章，你将了解受限玻尔兹曼机和深度信念网络算法的原理，学会如何使用它们，并且能够提升你所得到的结果的质量。本章内容总结如下所示：

- 神经网络入门
- 受限玻尔兹曼机
- 深度信念网络

2.1 神经网络入门

受限玻尔兹曼机是循环神经网络的一种形式。为了理解受限玻尔兹曼机的工作原理，必须先大体认识神经网络。如果对人工神经网络（后文简称神经网络）算法有所了解，那么你会在以下描述中发现熟悉的字眼。

许多高手都对神经网络进行了详细的理论描述，因此这里不会详细探讨。本章将先介绍神经

网络的组成部分、一般架构和常见的学习过程。

2.1.1　神经网络的组成

如果是初次学习神经网络，那么可将神经网络理解为一类数学模型，这类数学模型可根据一系列的输入特征，通过训练来迭代和优化定义的目标函数（或分布）。一般用运算符的形式定义模型的性能测量方法（通常为成本函数），并用于表示神经网络应用的具体目标，如此一来，神经网络就可以用来分类、预测或转换数据。

神经网络中的“神经”一词延续了传统，即借助笨重的生物学词汇进行比喻，以激励人们研究机器学习。因此，人工神经网络算法最初借鉴了（而且现在依然经常这么做）生物神经元的结构。

一个神经网络由以下元素组成。

- 一个学习过程：神经网络通过调整节点的权重函数的参数进行学习。这要求将性能测量方法（如前文所说，在监督学习的背景下，性能测量方法往往指成本函数，用于度量神经网络输出的精确程度）的输出项反馈到网络的学习函数中。然后学习函数会调整权重，以实现成本函数最小化（技术上来说，方式通常为计算函数的偏导数，这是梯度下降法中使用的一个术语）。
- 一组神经元或权重：每个神经元或权重都包含一个权重函数（即激活函数），用于处理输入数据。神经网络不同，激活函数也会有很大差异（双曲正切函数就是一个著名的例子）。权重必须具有适应性，即能够随学习过程的更新而调整。为了进行非参数建模（即无须定义概率分布的细节就能有效建模），必须同时使用可见单元和隐藏单元，其中隐藏单元无法被观察到。
- 连接功能：用于控制节点之间的数据传输。无论是否受限，节点都可以自由地将数据传输给另一节点，但在输入数据必须向指定方向传输的层中，节点结构会更复杂。互连模式是多样的，这丰富了其生成的网络的功能性和可能性。

这些元素允许我们构造各种各样的神经网络，从熟悉的有向非循环图（其中最有名的可能是**多层感知机**）到其他创新的种类。上一章中应用的**自组织映射**就是一种神经网络，其学习过程非常独特。本章即将深入探讨的受限玻尔兹曼机算法则是另一种神经网络算法，同样也拥有许多独有性质。

2.1.2　网络拓扑结构

神经元在神经网络中的连接方式有很多种，而结构将在很大程度上决定神经网络的学习能力。无监督学习中的常见拓扑结构与监督学习中的存在区别，你现在已经熟悉的一种常见无监督学习拓扑结构就是上一章探讨过的自组织映射。

如我们所见，自组织映射能直接将单个输入项映射到每个节点包含的权重向量，然后对节点重新排序，直到收敛到数据集的一个合理映射。受到具体训练细节的影响，自组织映射的实际结构会有许多种形式，一般取决于具体的训练数据实例和神经网络结构的设计，但是矩形和六边形的网格结构如今愈发普遍。

监督学习中一种非常常见的拓扑结构是三层前馈神经网络，其中最典型的一种就是多层感知机。在该网络拓扑模型中，网络中的神经元被分成几层，其中每一层都向“上一层”传递信息。第一层将模型输入提供给隐藏层。隐藏层用权重激活（利用合理的激活函数，如 sigmoid 函数或高斯函数，多层感知机可以用作全局函数逼近器）表示数据，然后将激活值传输到输出层。最后输出层会输出神经网络结果。这样的拓扑结构如图 2-1 所示。

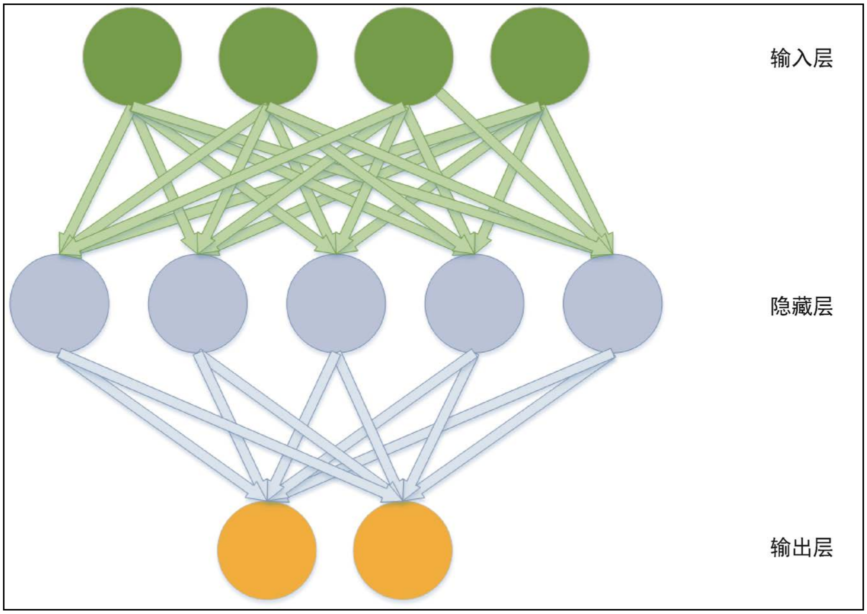


图　2-1

其他的网络拓扑结构则提供不同功能。比如，玻尔兹曼机的拓扑结构就与上文提到的拓扑结构不同。和普通的三层网络一样，它也包含隐藏神经元和可见神经元，但这些神经元两两相连，构成一幅有向循环图（见图 2-2）。

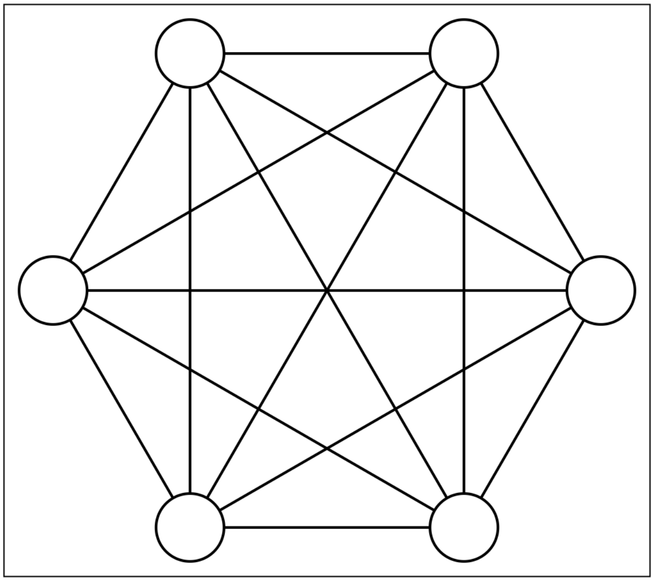


图　2-2

这样的拓扑结构使玻尔兹曼机具有随机性（遵从概率而非确定的），而且面对复杂问题时，其发展方式有许多种。玻尔兹曼机也具有生成性，即可以充分利用所有输入变量进行（概率分布上的）建模，而不是用观测变量针对目标变量进行特定建模。

哪种网络拓扑结构更合适，很大程度上取决于要处理的问题和所需要的输出结果。每种拓扑结构都有其使用的场景。而且本章所提及的每种拓扑结构都带有一种学习过程，这种学习过程能使网络迭代地收敛到（理想情况下最优的）解决方案。

学习过程多种多样，其中的具体过程和网络拓扑结构之间多少存在共通之处。学习过程的目的是迭代地调整神经网络的权重，逐渐精确地表示输入数据。

就像网络拓扑结构，学习过程的种类也有很多。有的你已经非常熟悉，而且有许多优质学习资源可供选择（本章末尾提供了许多精品案例）。本章将主要介绍学习过程的共同特征，并详细探讨一个具体案例。

我们已经说过，神经网络中学习的目的是迭代地改进整个模型的权重分布，以使其更精确地逼近输入数据背后的函数。该过程需要性能评价方法，可以是监督分类（即用多层感知机网络的反向传播学习算法）常用的分类误差测量。在随机网络中，也可以是将某一项概率最大化（比如基于能量的网络中的能量）。

无论是哪一种神经网络，只要有提升概率的指标，网络就能应用优化方法来有效改善该指标的度量。很多情况下，网络优化都是通过**梯度下降法**实现的。对于该方法来说，训练中每次迭代

时的评价指标值就好比梯度的斜率。因此，评价指标的最小化，就相当于将梯度下降到在对应权重下误差测度最小的点。

下一次迭代过程中网络更新的步长（即算法的学习率）既可以随评价指标变化而变化，也可以固化在代码中。

权重更新（网络据此调整）可能来源于误差表面本身；如果确实如此，那么网络往往能够计算梯度，并用于调整网络的激活权重函数的参数值，从而继续改善评价指标。

回顾了网络拓扑结构和学习方法的一般概念后，接下来开始探讨一种具体的神经网络——受限玻尔兹曼机。你将学习到，受限玻尔兹曼机是强大的深度学习算法中非常重要的一员。

2.2　受限玻尔兹曼机

受限玻尔兹曼机是本章主题深度学习架构（深度信念网络）的基础。接下来几节将介绍受限玻尔兹曼机背后的原理，包括其总体结构和学习过程。

然后，我们将直接研究受限玻尔兹曼机类的代码，将理论元素和代码中的函数联系起来。最后，我们将稍稍涉猎受限玻尔兹曼机的应用，以及与实现受限玻尔兹曼机相关的实际因素。

2.2.1　受限玻尔兹曼机简介

玻尔兹曼机是一种特殊的随机循环神经网络。它是一种基于能量的模型，也就是能够用能量函数将能量值和网络结构联系起来的模型。

前面简要探讨了玻尔兹曼机的结构。如前文所说，玻尔兹曼机是一种有向循环图，其中每个节点都与其他所有节点相连。该性质使得玻尔兹曼机能够递归建模，因此我们可以观测到模型输出是如何随着时间推移变化的。

玻尔兹曼机中的学习循环包括最大化训练数据集 X 的概率。前文提过，此处所使用的模型性能评价指标是能量，定义为给定模型参数向量 Θ 时数据集 X 的概率的负对数。计算该指标并用于更新网络权重，以便最小化网络中的自由能。

玻尔兹曼机尤其适用于图像数据处理，其中包括照片、面部特征和手写分类。

但玻尔兹曼机并不适合处理难度更大的机器学习问题，这是因为问题难度增大往往伴随着对计算力需求的增加。随着节点数增加，计算时间会呈指数级增长，最终将无法计算出网络中的自由能。

> 如果对背后的形式推理感兴趣，不妨了解一下：由于数据点 x 的概率密度函数 $p(x;\Theta)$对所有 x 的积分结果必须为 1，因此需要一个配分函数 Z 作为标准化常量。（Z 是常量，一个非负函数乘以 Z，可以使其在定义域内的积分为 1；本例中也就是对所有的 x 积分。）
>
> 概率模型函数是一系列正态分布的函数。要想知道模型的能量，必须区分模型的每个参数；然而，由于配分函数的存在，这一点变得非常困难。每个模型参数都会生成一些依赖于其他模型参数的方程，到最后，我们将不得不在（可能的）高成本计算的基础上计算出能量，而这样的计算成本会随网络规模的增大而变得愈发高昂。

为了克服玻尔兹曼机的这个缺点，必须调整网络拓扑结构和训练过程。

1. 拓扑结构

通过限制节点间的连接这一拓扑结构变换方式可以显著提高计算效率。首先，必须防止同一层的节点相互连接。此外，所有跨层连接（即不相邻层之间的直接连接）也是禁止的。这种结构的玻尔兹曼机就称为受限玻尔兹曼机，如图 2-3 所示。

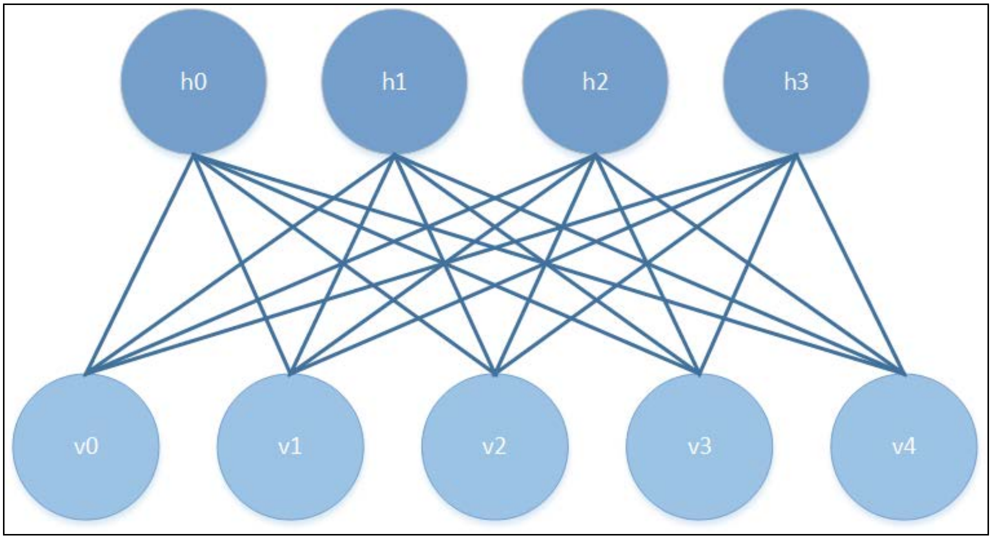


图　2-3

这种拓扑结构的一个优势是，隐藏层和可见层是互相条件独立的，这样我们就能根据一个层的激活值对另一层进行采样[①]。

① 这样做是为了得到条件概率的估计。——译者注

2. 训练过程

前文提到过，随着玻尔兹曼机的规模增大，其训练时间也会大幅增长，这使得我们最终无法计算训练阶段所用的能量函数。

受限玻尔兹曼机使用了一种本质上不同的学习算法——**永久对比散度**（permanent contrastive divergence，PCD）算法，该算法能生成极大似然概率的近似值。永久对比散度不会求出能量函数本身的值，但会估算能量函数的梯度值。因此，我们可以沿着最陡峭的梯度不断微调模型，并最终到达局部最优解。

永久对比散度算法由两阶段组成，分别称为正相位阶段和负相位阶段，每一阶段都对模型能量有相应的影响。正相位阶段会提升训练数据集 X 的概率，从而降低模型的能量。然后，负相位阶段会用模型的一种采样方法来估计负相位梯度。负相位阶段的整体效果是降低模型抽取的样本概率。

负相位阶段和整个更新过程的采样所采用的方法叫 Gibbs 采样。

> Gibbs 采样是**马尔可夫链蒙特卡洛**（Markov Chain Monte Carlo，MCMC）系列算法的变种，从近似的多元概率分布中采样。也就是说，该方法在构建概率模型的过程中没有使用加和计算（比如掷硬币一定次数时所做的计算，此时我们会将掷出正面的次数相加，再除以总次数），而是粗略估计积分值。由于篇幅所限，本书无法详述通过粗略估计积分值来构建概率模型的问题，但 2.4 节将提供一些优质的参考文献。目前需要谨记的关键（剔除许多重要细节）是，每个数据点不要只计算一次，而要基于数据的分布（通常是非均匀的分布）进行计算。基于模型中的其他所有参数值，Gibbs 采样对模型每个参数的概率采样，获得新参数值后，会立即将该值用于其他参数的采样计算。

你可能会问：为什么必须使用永久对比散度？为什么不用更熟悉的方法，比如包含线性搜索的梯度下降法？简单来说，我们没法轻易计算出网络中的自由能，因为该计算包含所有网络节点的集合。遇到玻尔兹曼机的巨大缺陷时，我们发现了这一局限——节点的增多会带来计算时间的指数级增长，导致最后无法计算出函数值，也就无法将函数最小化！

永久对比散度能估算能量函数的梯度，逼近网络的自由能，从而很快将估算结果用于实践，且该估算一般比较精确。（有关性能比较，参见 2.4 节。）

如前文所述，受限玻尔兹曼机的概率模型函数是模型参数的联合分布，这使得 Gibbs 采样更加精确。

初始化后的受限玻尔兹曼机的训练循环包含以下几步：

(1) 得到此次迭代的隐藏层的激活权重值；
(2) 将上次迭代的 Gibbs 链的状态作为输入项，执行永久对比散度正相位阶段；
(3) 利用已有的 Gibbs 链的状态执行永久对比散度负相位阶段，得到自由能值；
(4) 用计算出的能量值更新隐藏层的激活权重。

该算法让受限玻尔兹曼机能够在自由能值降低的方向上逐步迭代，直到训练集的概率积分为 1，自由能等于 0，此时受限玻尔兹曼机收敛，训练结束。

学习过受限玻尔兹曼机的拓扑结构和训练过程后，接下来我们用算法对一个大规模的真实数据集分类。

2.2.2　受限玻尔兹曼机的应用

现在我们已经对受限玻尔兹曼机算法的运行机制有所了解，下面来浏览一下构造受限玻尔兹曼机的代码。我们将用一个 `RBM` 类为 MNIST 手写数字数据集分类。我们要研究的代码将执行以下几步。

- 设定受限玻尔兹曼机的初始参数，包括层的尺寸、可共享的偏差向量以及可与外部网络结构连接的可共享的权重矩阵（以便应用深度信念网络）。
- 定义用于隐藏层和可见层之间信息交换和推理的函数。
- 定义用于更新网络节点的参数的函数。
- 定义学习过程中用于高效抽样的函数，用永久对比散度-k 加速采样（使得计算能在合理的时间内完成）。
- 定义用于计算模型自由能的函数（用于计算永久对比散度-k 更新的梯度）。
- 定义**伪似然度**（Pseudo-Likelihood，PL），用来代替实际对数似然值并指导超参数的选取。

先从考查 `RBM` 类开始：

```
class RBM(object):
    def __init__(
        self,
        input=None,
        n_visible=784,
        n_hidden=500,
        w=None,
        hbias=None,
        vbias=None,
        numpy_rng=None,
        theano_rng=None
    ):
```

首先需要构造的是 `RBM` 类，这个类包含了很多模型参数的定义，比如可见节点数和隐藏节点数（`n_visible` 和 `n_hidden`），以及用于调整受限玻尔兹曼机的推理功能和对比散度更新的

其他参数。

参数 w 可用作指向共享权重矩阵的指针。本章后面会介绍到，该参数与深度信念网络关系密切；在这样的结构下，需要在网络的不同部分之间共享权重矩阵。

参数 hbias 和 vbias 的用法相似，分别用作隐藏单元和可见单元共享的偏差向量的可选参考。两者同样用于深度信念网络。

参数 input 则确保受限玻尔兹曼机与图中其他元素保持由上至下的连接性，这使得连接受限玻尔兹曼机成为可能。

构建好 RBM 类后，接下来需要给这些参数赋值：

```
self.n_visible = n_visible
self.n_hidden = n_hidden

if numpy_rng is None:
    numpy_rng = numpy.random.RandomState(1234)

if theano_rng is None:
    theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
```

以上代码非常浅显易懂。我们设置了受限玻尔兹曼机的可见节点和隐藏节点，以及两个随机数生成器。参数 theano_rng 将在受限玻尔兹曼机的隐藏单元中采样：

```
if W is None:
    initial_W = numpy.asarray(
        numpy_rng.uniform(
            low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            size=(n_visible, n_hidden)
        ),
        dtype=theano.config.floatX
    )
```

这段代码改变了 W 的数据类型，使其能在 GPU 上运行。接下来，我们用 theano.shared 设置了共享变量，这使得变量存储下来并可以在相关函数之间共享。在这个例子中，我们创建的共享变量是权重向量（W）以及隐藏单元与可见单元的偏差变量（分别为 hbias 和 vbias）。用多个成分构建深度网络时，以下代码允许我们在网络的不同部分之间共享这些成分：

```
W = theano.shared(value=initial_W, name='W', borrow=True)

if hbias is None:
    hbias = theano.shared(
        value=numpy.zeros(
            n_hidden,
            dtype=theano.config.floatX
        ),
        name='hbias',
```

```
            borrow=True
        )

    if vbias is None:
        vbias = theano.shared(
            value=numpy.zeros(
                n_visible,
                dtype=theano.config.floatX
            ),
            name='vbias',
            borrow=True
        )
```

此时，我们已经可以对输入层进行初始化，如下所示：

```
self.input = input
if not input:
    self.input = T.matrix('input')

self.W = W
self.hbias = hbias
self.vbias = vbias
self.theano_rng = theano_rng
self.params = [self.W, self.hbias, self.vbias]
```

初始化过 input 层后，下一步便是构造一幅前文提过的符号图。为此，我们需要构造一些函数，以控制层间的传播和网络的激活计算操作：

```
def propup(self, vis):
    pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
    return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]


def propdown(self, hid):
    pre_sigmoid_activation = T.dot(hid, self.W.T) + self.vbias
    return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
```

以上两个函数将一层单元的激活值传到另一层。第一个函数将可见单元的激活值向上传到隐藏层，因此隐藏层能够基于采样后的可见单元计算激活值。第二个函数则正相反，它将隐藏层的激活值向下传到可见单元。

你可能会问：为什么要构造两个函数 propup 和 propdown？前面提过，永久对比散度只要求我们从隐藏单元采样。那么 propup 如何取值呢？

简而言之，需要从受限玻尔兹曼机采样以回顾其进展时，可见层的采样尤其有用。在大部分用受限玻尔兹曼机处理可视化数据的应用中，需要定期对选取采样后的可见层进行输出并且绘图，如图 2-4 所示。

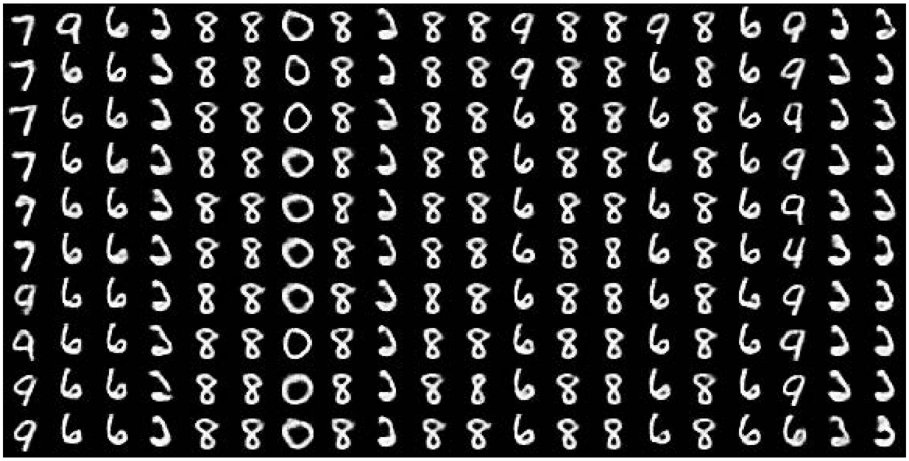


图 2-4

由图 2-4 可知，在迭代的过程中，我们的网络逐渐改变了标签；在第一个示例中，7 变成了 9，而在另一例中，9 变成了 6，最后网络的元素逐渐变成了 3。

我们曾探讨过，对受限玻尔兹曼机的运算检查次数越多越好，以确保能得到有意义的结果。对网络输出项进行采样就是提升可见性的一种方法。

学习过可见层激活的知识后，就可以在已知隐藏节点的激活的情况下，对隐藏层的单元激活进行采样：

```
def sample_h_given_v(self, v0_sample):

    pre_sigmoid_h1, h1_mean = self.propup(v0_sample)
        h1_sample = self.theano_rng.binomial(size=h1_mean.shape,
        n=1, p=h1_mean, dtype=theano.config.floatX)

        return [pre_sigmoid_h1, h1_mean, h1_sample]
```

同样，已知隐藏单元激活的情况下，也能对可见层采样：

```
def sample_v_given_h(self, h0_sample):
    pre_sigmoid_v1, v1_mean = self.propdown(h0_sample)
      v1_sample = self.theano_rng.binomial(size=v1_mean.shape,
                                           n=1, p=v1_mean,
                                           dtype=theano.config.floatX)

      return [pre_sigmoid_v1, v1_mean, v1_sample]
```

如本章前文所述，现在我们已经完成了网络内部的连接，以及执行 Gibbs 采样需要的更新循环。接下来该定义采样步骤了！

```
def gibbs_hvh(self, h0_sample):

    pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h0_sample)
    pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v1_sample)
    return [pre_sigmoid_v1, v1_mean, v1_sample,
            pre_sigmoid_h1, h1_mean, h1_sample]
```

如前所述，我们需要定义一个简单的函数来对可见层采样：

```
def gibbs_vhv(self, v0_sample):

    pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v0_sample)
    pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h1_sample)
    return [pre_sigmoid_h1, h1_mean, h1_sample,
            pre_sigmoid_v1, v1_mean, v1_sample]
```

目前编写的代码给出了模型的雏形，其中设置了节点和层，以及层之间的连接。该段代码可以基于对隐藏层的 Gibbs 采样更新网络。

但我们还缺少代码来实现以下步骤。

- ❑ 计算模型的自由能。如前所述，模型用能量来达成以下目的：
  - ■ 用 Gibbs 采样的相关代码完善永久对比散度，并设置 Gibbs 采样的计数参数 $k=1$，以计算梯度下降法的参数梯度；
  - ■ 构造一个方法，从而将永久对比散度的结果（计算的梯度值）传到先前定义的网络更新代码中。
- ❑ 构造方法来记录受限玻尔兹曼机的训练进度和结果。

首先，要构造计算受限玻尔兹曼机自由能的方法。注意，这是先前讨论过的隐藏层概率分布的逆对数：

```
def free_energy(self, v_sample):

    wx_b = T.dot(v_sample, self.W) + self.hbias
    vbias_term = T.dot(v_sample, self.vbias)
    hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
    return -hidden_term - vbias_term
```

然后，实现永久对比散度。这里我们要设置一些有趣的参数：`lr` 是学习率（learning rate）的缩写，它是用于调整学习速度的可调节参数；`k` 则指代永久对比散度执行的步骤数。（还记得前文提到的永久对比散度-k 吗？）

前面讨论过永久对比散度包含两个阶段，即正相位阶段和负相位阶段。以下代码计算了永久对比散度的正相位阶段：

```
def get_cost_updates(self, lr=0.1, persistent = , k=1):

    pre_sigmoid_ph, ph_mean, ph_sample = self.sample_h_given_v(self.input)



        chain_start = persistent
```

与此同时，下一段代码实现了永久对比散度的负相位阶段。我们用 Theano 包的 `scan` 函数循环执行 `gibbs_hvh` 函数 *k* 次，每次循环都要执行一次 Gibbs 采样。负相位阶段结束后，就可以得到自由能值：

```
(
    [
        pre_sigmoid_nvs,
        nv_means,
        nv_samples,
        pre_sigmoid_nhs,
        nh_means,
        nh_samples
    ],
    updates
) = theano.scan(
    self.gibbs_hvh,
    outputs_info=[None, None, None, None, None, chain_start],
    n_steps=k
)


chain_end = nv_samples[-1]

cost = T.mean(self.free_energy(self.input)) - T.mean(
    self.free_energy(chain_end))

gparams = T.grad(cost, self.params,
                 consider_constant=[chain_end])
```

写完执行整个永久对比散度过程的代码后，还需要一个将输出项传给网络的方法。此时可以将永久对比散度学习过程和代码相结合，以更新我们之前探讨过的网络。之前的字典（一种数据结构）`updates` 指向了 Theano 中执行 `gibbs_hvh` 函数的 `scan` 函数的结果。你可能还记得，`gibbs_hvh` 目前包含 `theano_rng` 的随机状态规则。我们目前要做的是，在字典（变量 `updates`）中添加包含 Gibbs 链状态的新参数值和变量：

```
for gparam, param in zip(gparams, self.params):
    updates[param] = param - gparam * T.cast(
        lr, dtype=theano.config.floatX
    )

    updates = nh_samples[-1]
    monitoring_cost = self.get_pseudo_likelihood_cost(updates)


return monitoring_cost, updates
```

现在几乎具备了让受限玻尔兹曼机运行的所有条件。但仍缺少一种检查训练的方法，无论是训练过程中还是训练过程结束后。我们需要确保受限玻尔兹曼机学习的是数据的恰当表示法。

前文讨论了如何训练受限玻尔兹曼机，特别提及了配分函数带来的困难。此外，在编写代码前期，我们实现了一种可以在训练过程中检查受限玻尔兹曼机的方法，还构造了 `gibbs_vhv` 函数来对模型进行 Gibbs 采样。

在前文讨论如何验证受限玻尔兹曼机时，我们探讨了如何将受限玻尔兹曼机创建的滤波器可视化。接下来简要回顾该实现。

最后的改进可能就是将最大似然估计替换为 PL 的逆对数，以便模型更易处理。从技术上来说，log-PL 是每个数据点（每个 x）在其他所有数据点的条件下的对数概率。如前所述，当数据集过大、维数过多时，这种计算的成本会过高，因此不妨使用 log-PL 的随机近似。

我们引用了一个能在 `get_cost_updates` 函数中计算 PL 成本的函数，即 `get_pseudo_likelihood_cost` 函数。现在我们来具体定义一下这个函数，并获取伪似然度：

```
def get_pseudo_likelihood_cost(self, updates):

    bit_i_idx = theano.shared(value=0, name='bit_i_idx')
    xi = T.round(self.input)

    fe_xi = self.free_energy(xi)

    xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:, bit_i_idx])

    fe_xi_flip = self.free_energy(xi_flip)

    cost = T.mean(self.n_visible * T.log(T.nnet.sigmoid(fe_xi_flip - fe_xi)))


    updates[bit_i_idx] = (bit_i_idx + 1) % self.n_visible

    return cost
```

现在补上了所有缺失的部分，而且完整回顾了 `RBM` 类。我们探究了每个元素和受限玻尔兹曼机背后理论的联系。至此，你应该已经全面了解了受限玻尔兹曼机算法的运行原理。我们知道了受限玻尔兹曼机的输出项是什么，并且很快就能够检查和评估模型了。简而言之，我们已经做好训练受限玻尔兹曼机的准备了。训练受限玻尔兹曼机需要运行以下代码。代码中涉及了 `train_set_x` 函数，本章后文将详细讨论该函数。

```
train_rbm = theano.function(
    [index],
    cost,
    updates=updates,
    givens={
        x: train_set_x[index * batch_size: (index + 1) * batch_size]
```

```
        },
        name='train_rbm'
    )

    plotting_time = 0.
    start_time = time.clock()
```

更新过受限玻尔兹曼机的 `updates` 变量和训练集后，现在可以开始训练了。在每次迭代中，先训练数据，然后将权重绘制为矩阵（如本章前文所述）：


```
    for epoch in xrange(training_epochs):

        mean_cost = []
        for batch_index in xrange(n_train_batches):
            mean_cost += [train_rbm(batch_index)]

        print 'Training epoch %d, cost is ' % epoch,
        numpy.mean(mean_cost)

        plotting_start = time.clock()
        image = Image.fromarray(
            tile_raster_images(
                X=rbm.W.get_value(borrow=True).T,
                img_shape=(28, 28),
                tile_shape=(10, 10),
                tile_spacing=(1, 1)
            )
        )
        image.save('filters_at_epoch_%i.png' % epoch)
        plotting_stop = time.clock()
        plotting_time += (plotting_stop - plotting_start)

    end_time = time.clock()

    pretraining_time = (end_time - start_time) - plotting_time

    print ('Training took %f minutes' % (pretraining_time / 60.))
```

权重的图像一般比较容易识别，而且会很像 Gabor 滤波器（常用于图像边缘检测的线性滤波器）。如果数据集是在噪声极低的背景上的手写数字，你会发现，权重（的图像）能体现笔触的痕迹。而对照片数据集来说，滤波器将近似地追踪图像的边缘。图 2-5 就是一个输出示例。

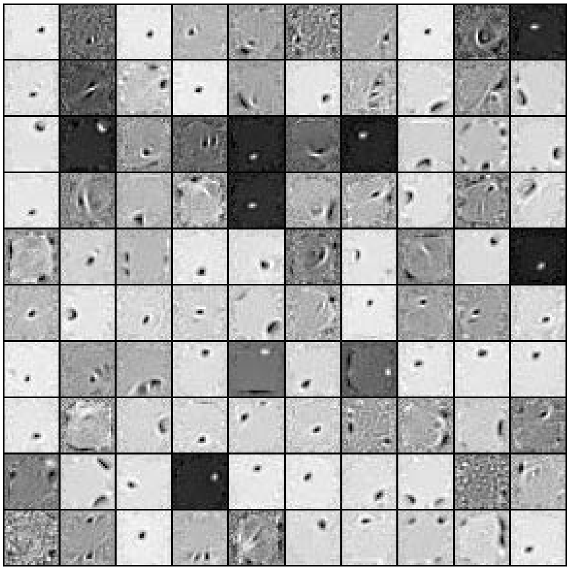


图　2-5

最后，我们构造了一个用于获取采样的持续 Gibbs 链。如前所述，以下函数执行了一步 Gibbs，然后更新了整条链：

```
plot_every = 1000

    (
        [
            presig_hids,
            hid_mfs,
            hid_samples,
            presig_vis,
            vis_mfs,
            vis_samples
        ],
        updates
    ) = theano.scan(
        rbm.gibbs_vhv,
        outputs_info=[None, None, None, None, None, persistent_vis_chain],
        n_steps=plot_every
    )
```

这段代码运行了之前提到的 gibbs_vhv 函数，来对抽样后的网络输出进行绘图，以便检查：

```
updates.update({persistent_vis_chain: vis_samples[-1]})
sample_fn = theano.function(
    [],
    [
        vis_mfs[-1],
        vis_samples[-1]
```

```
        ],
        updates=updates,
        name='sample_fn'
    )

    image_data = numpy.zeros(
        (29 * n_samples + 1, 29 * n_chains - 1),
        dtype='uint8'
    )
    for idx in xrange(n_samples):

        vis_mf, vis_sample = sample_fn()
        print ' ... plotting sample ', idx
        image_data[29 * idx:29 * idx + 28, :] = tile_raster_images(
            X=vis_mf,
            img_shape=(28, 28),
            tile_shape=(1, n_chains),
            tile_spacing=(1, 1)
        )

    image = Image.fromarray(image_data)
    image.save('samples.png')
```

至此，受限玻尔兹曼机已经完整。我们有了永久对比散度算法，也能用该算法以及 Gibbs 采样来更新网络，还有许多可视化输出方法，因此能够评估受限玻尔兹曼机的训练效果。

但还没结束！接下来，我们将介绍最常见、最强大的受限玻尔兹曼机应用。

2.2.3 受限玻尔兹曼机的扩展应用

我们既可以将受限玻尔兹曼机本身作为机器学习算法使用，也可以将其很好地与其他算法结合使用。其优势在于能扩充至对高维数据集的学习。但这并不是受限玻尔兹曼机的真正强项。

受限玻尔兹曼机最常见的应用是，作为称作深度信念网络的高效深度网络架构的预训练机制。深度信念网络是一种可用来学习和分类一系列图像数据集的强大工具，它的泛化能力是众多图像学习工具中最好的。因此，它被许多世界顶级的技术公司和数据科学企业采用，尤其是图像搜索和图像识别领域。

2.3 深度信念网络

深度信念网络是一种使用多个堆叠受限玻尔兹曼机构成的图模型。其中第一个受限玻尔兹曼机基于训练数据的像素点训练特征层，接下来的层则将上一层的激活值作为像素点（输入），并通过之后的隐藏层学习（更多的）特征。该过程通常称为学习数据的表示，这是深度学习中的一个常见主题。

所需要的受限玻尔兹曼机的数量取决于待处理问题的需求。从实际角度看，就是在精确度和

计算成本之间进行权衡。每个受限玻尔兹曼机层都会提高训练数据对数概率的下限。换句话说，随着层数的增加，深度信念网络会稳定减少犯错误的可能。

在层的尺寸方面，一般来说，最好减少相继受限玻尔兹曼机的隐藏层中的节点数。尽量避免受限玻尔兹曼机的可见单元数达到或超过上一层受限玻尔兹曼机的隐藏单元数（因为这样容易增加网络的识别风险）。

比较好的（但绝非必须）情况是，相继受限玻尔兹曼机的层逐渐缩小，直到最后一层受限玻尔兹曼机的层约等于数据中方差的维度数。在节点数过多的深度信念网络底层附加多层感知机会影响分类效果，就好像在水管末端接了一截吸管一样。多层感知机的神经元再多，训练也无法取得成功。另一方面，即使层包含的节点数不多，但层数够多，那么建模或多或少都是有效的。

决定数据中方差的维度数并非易事。上一章中介绍过的辅助工具——主成分分析能帮我们大致了解输入数据中存在多少大小有意义的成分。

2.3.1　训练深度信念网络

训练深度信念网络的过程通常基于贪心算法，也就是说，训练的目标是每层最优，而非全局最优。其学习过程如下所示。

- 我们用前面有关受限玻尔兹曼机学习的讨论中涉及的方法来训练深度信念网络的第一层。对隐藏单元进行 Gibbs 采样，将第一层的数据分布转化成后验分布。
- 现在的分布比输入数据的分布更有助于受限玻尔兹曼机训练，因此，下一个受限玻尔兹曼机层可以学习这个分布了。
- 后续的受限玻尔兹曼机层继续用上一层的输出进行训练。
- 使用一种性能评价指标调整架构中的所有参数。

性能评价指标有多种。如本章前面所说，它可以是梯度下降法用到的对数似然替代指标。在监督学习中，可以将分类器（如多层感知机）作为架构的最后一层，并将预测精度作为性能评价指标来进行深度网络架构的微调。

接下来我们将实际应用深度信念网络。

2.3.2　应用深度信念网络

讨论完深度信念网络及其相关理论后，现在我们可以构建自己的深度信念网络了。与探讨受限玻尔兹曼机时相似，我们先定义一个 `DBN` 类，然后将理论与代码相结合，讨论该网络可以做什么以及如何评价网络的性能，然后进行初始化、训练网络，并查看结果。

我们先看看 `DBN` 类：

```
class DBN(object):

    def __init__(self, numpy_rng, theano_rng=None, n_ins=784,
                 hidden_layers_sizes=[500, 500], n_outs=10):

    self.sigmoid_layers = []
    self.rbm_layers = []
    self.params = []
    self.n_layers = len(hidden_layers_sizes)

    assert self.n_layers > 0

    if not theano_rng:
        theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))


    self.x = T.matrix('x')
    self.y = T.ivector('y')
```

DBN 类包含了一些参数，其含义需要进一步解释。numpy_rng 和 theano_rng 用于定义初始权重，学习 RBM 类时已经介绍过了。n_ins 是指向深度信念网络输入数据维度（特征个数）的指针。hidden_layers_sizes 是记录隐藏层每层尺寸的列表，列表中的每个值会指导 DBN 类构造对应尺寸的受限玻尔兹曼机层。你可能会发现，n_layers 指的是网络中的层数，并且由 hidden_layers_sizes 决定。调整该列表中的值可以创建深度信念网络，其层尺寸从输入层起逐渐缩小，对数据的表示越来越简明，如前所述。

同样值得注意的是，self.sigmoid_layers 会存储多层感知机的成分（深度信念网络的最后一层），而 self.rbm_layers 则存储受限玻尔兹曼机层并用于预训练多层感知机。

完成后，按以下步骤完善深度信念网络架构：

- 构造 n_layers 个 sigmoid 层；
- 将 sigmoid 层相连，形成多层感知机；
- 利用共享权重矩阵为每个 sigmoid 层构建受限玻尔兹曼机，并在 sigmoid 层和受限玻尔兹曼机之间构建偏差项。

以下代码用 sigmoid 激活函数构建了 n_layers 个层，首先是输入层，然后构造了尺寸与 hidden_layers_sizes 列表中的值相对应的隐藏层：

```
for i in xrange(self.n_layers):

    if i == 0:
        input_size = n_ins
    else:
        input_size = hidden_layers_sizes[i - 1]
    if i == 0:
        layer_input = self.x
    else:
```

```
        layer_input = self.sigmoid_layers[-1].output

        sigmoid_layer = HiddenLayer(rng=numpy_rng,
                                    input=layer_input,
                                    n_in=input_size,
                                    n_out=hidden_layers_sizes[i],
                                    activation=T.nnet.sigmoid)
        self.sigmoid_layers.append(sigmoid_layer)

        self.params.extend(sigmoid_layer.params)
```

然后，构造一个与 sigmoid 层共享权重的受限玻尔兹曼机。这一步直接利用了之前所说的 RBM 类：

```
rbm_layer = RBM(numpy_rng=numpy_rng,
                theano_rng=theano_rng,
                input=layer_input,
                n_visible=input_size,
                n_hidden=hidden_layers_sizes[i],
                W=sigmoid_layer.W,
                hbias=sigmoid_layer.b)
self.rbm_layers.append(rbm_layer)
```

最后，在深度信念网络的末尾添加一个逻辑回归层，以形成多层感知机：

```
self.logLayer = LogisticRegression(
    input=self.sigmoid_layers[-1].output,
    n_in=hidden_layers_sizes[-1],
    n_out=n_outs)
self.params.extend(self.logLayer.params)

self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)

self.errors = self.logLayer.errors(self.y)
```

组合好 MLP 类后，就可以构建 DBN 了。以下代码用 28 * 28 个输入项（即 MNIST 图像数据中的 28×28 个像素）、3 个尺寸逐渐缩小的隐藏层和 10 个输出值（MNIST 数据集的 10 个手写数字类）构造了网络。

```
numpy_rng = numpy.random.RandomState(123)
print '... building the model'
dbn = DBN(numpy_rng=numpy_rng, n_ins=28 * 28,
          hidden_layers_sizes=[1000, 800, 720],
          n_outs=10)
```

本节前面说过，深度信念网络的训练分为两个阶段——每个层用上一层的输出项进行训练的逐层预训练阶段，以及允许整个网络调整权重的细调阶段（反向传播阶段）。第一步的预训练要将永久对比散度的一个步骤用于每层的受限玻尔兹曼机，如下所示：

```
print '... getting the pretraining functions'
pretraining_fns = dbn.pretraining_functions(train_set_x=train_set_x,
                                            batch_size=batch_size, k=k)
```

```
print '... pre-training the model'
start_time = time.clock()

for i in xrange(dbn.n_layers):
    for epoch in xrange(pretraining_epochs):
        c = []
        for batch_index in xrange(n_train_batches):
            c.append(pretraining_fns[i](index=batch_index,
                                        lr=pretrain_lr))
        print 'Pre-training layer %i, epoch %d, cost ' % (i, epoch),
        print numpy.mean(c)

end_time = time.clock()
```


运行预训练后的深度信念网络则可以通过以下命令来实现：

```
python code/DBN.py
```

注意，即使有 GPU 的加速，预处理也要花很长时间，因此建议你在夜间运行这段代码。

2.3.3 验证深度信念网络

我们非常熟悉验证深度信念网络的整个过程。可以将交叉验证的最小验证误差作为一种误差测度，但最小交叉验证误差会低估交叉验证数据的误差，因为元参数可能会过度拟合新数据。

因此，应该先用交叉验证误差来调整元参数，直到交叉验证误差达到最小。然后将深度信念网络应用于保留的测试集，将测试误差作为验证测度。DBN 类执行的就是如上训练过程。

然而，我们仍然不知道网络的训练是否充分。如果深度信念网络的表现不够理想，该怎么办呢？

首先要识别潜在原因。存在很多常见原因。我们知道，网络背后的受限玻尔兹曼机的训练是很棘手的，其中每个单独层的训练都有可能遭遇失败。好在 RBM 类让我们能够看到每个层生成的权重（滤波器），而且我们还可以将权重可视化，以了解网络到底想要表示什么。

此外，我们还想知道网络是否出现了过度拟合或欠拟合。这两种情况都可能发生，我们需要识别情况的严重程度及其发生的原因。在欠拟合的情况下，训练过程可能单纯找不到适合模型的参数，这在使用更大型的网络解决大型问题空间时尤其常见，但也可能发生在较小的模型上。如果你的深度信念网络出现了欠拟合问题，可以选择以下两种方法。一种是单纯缩小隐藏层的尺寸，这种方法可能成功，也可能失败。另一种更好的选择是逐渐减少隐藏层，这样每个层都能比之前的层学习到更精炼的数据表示。至于如何做、减少多少、何时停止，都与第一种情况的反复试验和基于经验的长期学习有关。

过度拟合也是一种众所周知的现象，它出现在算法对训练数据过度训练时。这类问题相当有

害，通常可在交叉验证中被识别出来（此时错误率会大幅升高）。解决过度拟合问题的方法的确存在，比如可以增大训练数据集的规模。另一种更笨拙的贝叶斯方法是，附加一个条件（如某种先验分布），以降低训练数据的拟合度。提升分类效果的最有效方法是预处理方法，第 6~7 章将详细探讨该问题。

虽然这段代码能从预定义的情况下（给定种子值）初始化，但模型的随机本质仍会导致其很快出现偏离，因此其结果是多样的。在我的系统上运行深度信念网络时，最小交叉验证误差是 1.19%。更重要的是，经过 46 次监督学习后，测试误差仅为 1.30%。这些结果都不错，甚至能和行业领先的案例相媲美！

2.4　扩展阅读

若想阅读神经网络的入门读物，可以从多个渠道获取。需要考虑的注意事项有很多，不同作者会侧重不同主题。Kevin Gurney 撰写的“An Introduction to Neural Networks”提供了完整的介绍。

若需要马尔可夫蒙特卡洛的优质资料，参见 http://twiecki.github.io/blog/2015/11/10/mcmc-sampling/。

如果对 Gibbs 采样原理感兴趣，可以阅读 Philip Resnik 和 Eric Hardisty 的论文“Gibbs Sampling for the Uninitiated”。这篇文章的技术性比较强，但清晰明了地描述了 Gibbs 的运作方式，而且值得称赞的是其中有一些非常棒的类比。

对比散度的高质量讲解并不多，我比较喜欢的一篇文章是 Oliver Woodford 写的“Notes on Contrastive Divergence”。如果你被过多公式表达困扰，不妨阅读一下这篇文章，其中的理论解释很清晰，而且包含实际问题。

本章使用的 Theano 文档可在 http://deeplearning.net/tutorial/contents.html 上找到，这是受限玻尔兹曼机和深度信念网络课程讨论和实现的大本营。

2.5　小结

本章介绍了不少内容！首先对神经网络做了综述，主要关注了拓扑学和学习方法的一般性质，然后深入研究了受限玻尔兹曼机算法和受限玻尔兹曼机代码本身。扎实掌握这些知识后，着手构造了深度信念网络，并将深度信念网络的理论和代码相结合，然后在 MNIST 数据集上应用深度信念网络。我们在一个 10 类问题中应用了图像分类，并取得了非常优秀的成绩——分类误差竟低于 2%！

下一章将引入另一种深度学习架构——**堆叠式降噪自编码机**（stacked denoising autoencoder，SdA），你将继续增强深度学习本领。

第 3 章

堆叠式降噪自编码机

本章通过应用**堆叠式降噪自编码机**来介绍高维输入数据的特征表示，继续构建深度架构。

一如既往，本章首先阐释自编码机的理论和概念，详细介绍相关技术，以及如何将自编码机的优势融入数据科学工具组件，然后探讨**降噪自编码机**（denoising autoencoder，dA）的用途。它是一种算法的变体，这种算法能向输入数据人工引入随机的损坏成分，并通过自编码机修复损坏的数据，从而构建更有效的特征表示。

同样，介绍完基础理论之后，探讨降噪自编码机类的代码，并将理论和实现细节相结合，带领读者深入探究该技术。

流程与上一章相似：使用堆叠降噪自编码机构造一个深度架构，该架构能预训练多层感知机网络，还能提升许多无监督学习应用（包括语音数据处理）的性能。

3.1 自编码机

自编码机（也称 Diabolo 网络）是深度架构的一个重要组件。其训练方式和受限玻尔兹曼机类似，因此与之有相通之处。但自编码机的训练比包含对比散度的受限玻尔兹曼机的训练更简单，因此在后者的训练不太有效时，自编码机往往表现更佳。

3.1.1 自编码机简介

自编码机是一种简单的 3 层神经网络，其输出单元直接连接回输入单元。自编码机的目标是将 *i* 维输入项编码为 *h* 维的表示（$h<i$），然后在输出层重建（解码）输入数据。其训练过程包括迭代该过程，直到重建损失（也称“重构误差”）达到最小，此时应该能得到输入数据的最高效表示（排除了达到局部最小值的可能）。

前面介绍过一种强大的降维技术——主成分分析，而自编码机是最高效的输入数据降维表示法，两者毫无疑问非常相似，读者可能会感到困惑，为什么要介绍另一种用途相同的技术呢？

答案很简单，类似于自组织映射，自编码机能实现非线性降维，因此与主成分分析相比，处

理高维输入数据更高效。这又会引发疑问：既然自编码机和自组织映射的用途相同，又没有启发性的可视化表示，为什么要学习自编码机呢?

简而言之，自编码机更成熟、更复杂，降噪和堆叠技术能对高维、多模的数据降维，使得其训练比第 1 章介绍的技术更简单、精度更高、适用范围更广。

介绍过自编码机的性能后，下面详细介绍其拓扑结构和训练过程。

1. 拓扑结构

前面讲过，自编码机的结构相对简单。它是一个 3 层网络，分别为**输入**层、**隐藏**层和**输出**层。**输入**层前馈到**隐藏**层，然后是**输出**层，这与绝大多数的神经网络架构相同。一个值得一提的拓扑特征是，**隐藏**层的节点往往比**输入**层和**输出**层的更少。（但前文讲过，所需的**隐藏**节点数量其实是**输入**数据复杂度的函数，**隐藏**层的作用是限制**输入**层的信息内容，并迫使网络识别表示，这种表示可以捕获潜在的统计性质以用于数据。精确表示非常复杂的输入项需要大量隐藏节点。）

自编码机的关键特征是，**输出**层往往要设置为**输入**层的内容。自编码机的性能测度是，在其隐藏层内编码输入层，然后重建输入层的准确度。自编码机的拓扑结构通常如图 3-1 所示。

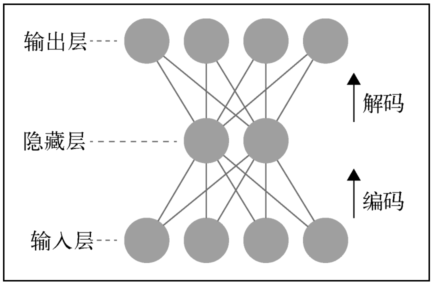


图　3-1

输入层和隐藏层之间的编码函数将输入（x）映射为新形式（y）。下面这个输入的非线性函数（本例中指 sigmoid 函数，s）就是一个简单的映射函数。

$$y = s(Wx + b)$$

然而，可能存在或需要设置更复杂的编码函数以适应特定对象领域。本例中的 W 表示赋给 x 的权重值，b 是一个可调变量，可随重建损失的最小化进程而调节。

然后，自编码机通过解码输出结果。重建是为了取得和 x 相同的结构，转换过程如下所示。

$$z = s(W'y + b')$$

其中，为了使网络达到最优，b'和 W'通常也是可配置的。

2. 训练过程

如前所述，网络通过最小化重建损失来进行训练。度量损失的一个常用方法是简单平方误差，如以下公式所示。

$$E=\frac{1}{2}\left\|z-x\right\|^{2}$$

如果输入项的形式比较特殊（比如一组比特概率），也可以应用其他更适合的误差评估指标。

虽然目标是让自编码机捕捉输入数据集中的主要变化，但自编码机仍可学习一些不那么有用的信息，比如输入项的恒等函数。

3.1.2 降噪自编码机

虽然自编码机在许多场景中表现出色，但并不适用于输入数据的分布复杂且必须使用高维模型建模的情况。主要问题是，当自编码机的输入层为*n*维，而编码后的数据维度[1]至少为*n*时，自编码机很可能只学习了输入项的恒等函数。此时，编码其实就是对输入层的复制。我们称这样的自编码机**过完备**。

> 训练机器学习技术时，必须理解隐藏层的维度是如何影响模型的质量的。当输入数据比较复杂，而隐藏层节点过少时，模型就无法有效地捕捉数据特征。显然，这样的网络不如节点更多的网络的训练效果好。
>
> 要想捕获输入数据的复杂分布，最好增加隐藏节点数。当隐藏层的节点数至少与输入层的相同时，网络很可能只学到了输入层的恒等函数。此时，输入层的每个元素都当作具体的独特案例来学习了。这样的模型对训练数据的模拟效果当然非常好，但它学习的是一种简单模式，无法泛化到其他数据集，很可能在验证时遭遇灾难性的失败。

该现象在对复杂数据（比如语音数据）建模时尤为普遍。这样的数据的分布往往非常复杂，因此语音信号的分类要求多模型编码，隐藏层的维度也应该较高。这显然也会增加自编码机（或大量模型，因为这不是自编码机独有的问题）学习恒等函数的风险。

虽然（很令人意外）过完备自编码机的确可以在某些情况下（例如第一个隐藏层需要非常小的权重，以确保隐藏单元线性排列且后续权重值较大）学习误差最小化表示法，但很难针对这种情况进行优化，因此需要用其他方法防止过完备自编码机学习恒等函数。

有很多方法能防止过完备自编码机学习恒等函数，并确保其在数据的表示中仍能捕获有效信

① 即“隐藏层维度”。——译者注

息。目前最常用的方法是在输入数据中引入噪声，迫使自编码机训练噪声数据，学习其分布和统计规律性，而不是恒等函数。许多方法能高效地完成这项工作，比如“稀疏约束”或 dropout 技术（其中输入值将随机设置为 0）。

本章将用 dropout 技术为输入数据引入噪声。通过这种方法，近半数的输入项将随机设置为 0。我们将构造一个随机扰动过程来干预输入数据，代码如下所示。

```
def get_corrupted_input(self, input, corruption_level):

    return self.theano_rng.binomial(size=input.shape, n=1, p=1 - corruption_level,
                                    dtype=theano.config.floatX) * input
```

为了准确地对输入数据建模，自编码机必须通过损坏数据来预测未损坏数据，然后学习有用的统计性质（即分布）。

除了能防止自编码机学习数据的恒等值，添加降噪过程还能使模型面对输入的变动或失真时表现得更加稳健。该过程尤其适用于噪声较多的输入数据，比如语音数据或图像数据。前言提到了，深度学习技术的显著优势之一是，其算法能将特征工程的需求最小化。许多学习算法能重建降噪输入数据并将其用于模型训练，但需要对输入数据进行冗长而复杂的预处理（比如图像信号滤波或音频信号处理），而降噪自编码机只需要极少的预处理步骤便能高效完成上述工作，从而大幅节约同等准确率下的训练时间。

最后，请注意，学习输入数据的恒等函数的自编码机可能会出现根本上的配置错误。由于自编码机有寻找特征集的低维表示的额外功能，学习了输入数据的恒等函数的自编码机可能单纯是因为节点过多。如有疑问，不妨试试减少隐藏层节点。

拓扑结构使得模型可以高效训练数据并提升自编码机的降噪功能。介绍完该结构后，下面将理论付诸实践，看看降噪自编码机的 Theano 代码。

3.1.3　应用降噪自编码机

下面介绍降噪自编码机的实现。再次强调，这次还是用 Theano 库来应用一个 `dA` 类。

与上一章的 `RBM` 类不同，降噪自编码机相对简单，而将降噪自编码机的功能与本章前面介绍的理论和数学知识联系起来也很简单。

第 2 章应用了一个 `RBM` 类，虽然其中包含的元素不是受限玻尔兹曼机正常运作所必需的，但这让多层深度架构共享参数成为可能。下面将使用的 `dA` 类能处理相似的共享元素，利用这些共享元素可以构建多层自编码机架构，稍后详述。

首先初始化一个 `dA` 类。指定可见单元的数量（`n_visible`）和隐藏单元的数量（`n_hidden`）。指定输入的配置（`input`）、权重（`W`）、隐藏偏差值和可见偏差值（分别为 `bhid` 和 `bvis`）。这 4

个额外的变量能使自编码机接收深度架构中其他元素的配置参数。

```
class dA(object):

    def __init__(
        self,
        numpy_rng,
        theano_rng=None,
        input=None,
        n_visible=784,
        n_hidden=500,
        W=None,
        bhid=None,
        bvis=None
    ):

        self.n_visible = n_visible
        self.n_hidden = n_hidden
```

然后初始化权重变量和偏差变量。给权重向量 ***W*** 赋一个初始值 `initial_W`，该值是在取值范围内随机均匀抽样得到的。

$$-4*\sqrt{\frac{6.}{(n_\text{hidden}+n_\text{visible})}} \quad to \quad 4*\sqrt{\frac{6.}{(n_{\text{hidden}}+n_{\text{visible}})}}$$

然后用 `numpy.zeros` 将可见偏差变量和隐藏偏差变量设置为 0 的数组。

```
if not theano_rng:
    theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))

if not W:
    initial_W = numpy.asarray(
        numpy_rng.uniform(
            low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
            size=(n_visible, n_hidden)
        ),
        dtype=theano.config.floatX
    )
    W = theano.shared(value=initial_W, name='W', borrow=True)

if not bvis:
    bvis = theano.shared(
        value=numpy.zeros(
            n_visible,
            dtype=theano.config.floatX
        ),
        borrow=True
    )

if not bhid:
    bhid = theano.shared(
```

```
            value=numpy.zeros(
                n_hidden,
                dtype=theano.config.floatX
            ),
            name='b',
            borrow=True
        )
```

如前所述，自编码机可以用 $y=s(\boldsymbol{W}x+b)$等映射来进行可见层和隐藏层的转换。为了实现这样的转换，必须先定义 `W`、`b`、`W'`和 `b'`，这些参数与前面介绍的自编码机参数 `bhid`、`bvis` 和 `W` 相关，其中 `W'`和 `b'`在以下代码中写作 `W_prime` 和 `b_prime`。

```
self.W = W
self.b = bhid
self.b_prime = bvis
self.W_prime = self.W.T
self.theano_rng = theano_rng
if input is None:
    self.x = T.dmatrix(name='input')
else:
    self.x = input

self.params = [self.W, self.b, self.b_prime]
```

这段代码分别将 `b` 和 `b_prime` 设置为 bhid 和 bvis，将 `W_prime` 设置为 `W` 的转置，也就是说，这些权重互相关联。自编码机有时（并非总是）会使用绑定权重（也称“受限权重”），原因如下。

- 很多时候绑定权重能提升结果的质量（虽然这些情况下的最佳方法是主成分分析，但带有关联权重的自编码机也能达到同等效果）。
- 绑定权重能减少需要存储的参数数量，从而降低自编码机的内存消耗。
- 最重要的是，绑定权重能起到正则化作用，需要优化的参数少了一个。（可能出错的事情也少了一件！）

但在其他情况下，更常用且更合适的还是未绑定（未受限）权重。的确，如果输入数据是多模的，而且最优解码器是对一系列非线性统计规律进行建模，线性模型（比如主成分分析）就不能有效地模拟出非线性趋势，此时未绑定权重反而效果更佳。

设置好自编码机的参数后，下面定义学习用的函数。前面讲过，通过向输入数据中添加噪声，自编码机可以有效地学习输入数据编码后的数据表示，这种数据表示可以反过来重建为输入数据。下面编写该功能对应的函数。首先损坏输入数据。

```
def get_corrupted_input(self, input, corruption_level):
    return self.theano_rng.binomial(size=input.shape, n=1, p=1 - corruption_level,
                                    dtype=theano.config.floatX) * input
```

损坏的程度由参数 `corruption_level` 设置。前面讲过，dropout 技术对输入数据施加的损

坏通常不会超过 50%（0.5）。该函数按照与输入数目成正比（比例等于 corruption_ level）的数量随机抽取实例，生成长度等同于输入数据的损坏向量，其分量为一系列 0 和 1，其中 corruption_level 个分量为 0。损坏的输入向量就是自编码机输入向量和损坏向量的简单乘积。

```
def get_hidden_values(self, input):
    return T.nnet.sigmoid(T.dot(input, self.W) + self.b)
```

用代码实现公式 $y=s(\boldsymbol{W}x+b)$并获取隐藏值 y。为获取自编码机的输出值（z），需要再写一段代码来重建隐藏层，将前面定义的 b_prime 和 W_prime 代入公式 $z = s(\boldsymbol{W}'y+b')$。

```
defget_reconstructed_input(self, hidden):
    returnT.nnet.sigmoid(T.dot(hidden, self.W_prime) + self.b_prime)
```


最后，还需要计算成本更新。前面回顾了一个简单的成本函数，即简单平方误差测量（$E=\frac{1}{2}\|z-x\|^2$）。下面将输入项（x）和重建输出项（z）代入该成本函数来计算成本更新。

```
def get_cost_updates(self, corruption_level, learning_rate):

    tilde_x = self.get_corrupted_input(self.x, corruption_level)
    y = self.get_hidden_values(tilde_x)
    z = self.get_reconstructed_input(y)
    E = (0.5 * (T.z – T.self.x)) ^ 2
    cost = T.mean(E)

    gparams = T.grad(cost, self.params)
    updates = [
        (param, param - learning_rate * gparam)
        for param, gparam in zip(self.params, gparams)
    ]

return (cost, updates)
```

这样就得到可以运行的降噪自编码机了。用它可以对输入数据的非线性性质进行建模，以及学习输入数据的有效低维表示。但自编码机的强大之处其实体现在多个自编码机堆叠时，这是深度架构的基本构成要素。

3.2 堆叠式降噪自编码机

虽然自编码机本身即是实用工具，但可以堆叠自编码机来构造深度网络，以达到更高的准确度。为此，需要将一层编码机生成的表示传给下一层编码机，作为该层的输入项。

目前许多顶尖数据科学团队使用**堆叠式降噪自编码机**（stacked denoising autoencoder，SdA）进行复杂的自然语言分析以及信号、图像和文本分析等。

堆叠式降噪自编码机的实现与上一章的深度信念网络非常相似。堆叠式降噪自编码机的使用也与深度信念网络中的受限玻尔兹曼机的使用大同小异。深度架构的每一层都包含一个降噪自编

码机和 sigmoid，前者用于预训练后者。堆叠式降噪自编码机的性能测度是训练集误差，包含一个层对层（逐层）预训练的高强度阶段，能在最后的微调阶段前逐渐匹配网络参数。在微调过程中，使用验证集和测试集来训练网络，更新迭代更少，但更新步长更大。训练的目标是使网络能在微调结束时收敛，以得到准确结果。

除了深度网络的典型优势（能学习复杂或高维数据集的特征表示，以及无须大量特征工程就能训练模型），堆叠式自编码机还有其他一些有趣的性质。

正确配置的堆叠式自编码机能捕获输入数据的**层级分组**（hierarchical grouping）。堆叠式降噪自编码机的连续层可以学习更高级的特征。第一层可能只学到输入数据中的一些一阶特征（比如照片中的边缘），第二层可以学到一阶特征的一些组合（比如通过结合轮廓信息和结构信息来学习特定部位的边缘）。

层数和层大小的确定没有固定法则。建议多试几个模型参数取值，以寻找最佳组合。试验时，最好选用超参数优化技术或遗传算法（稍后介绍）。

更高的层可以学习更高阶的特征，使得堆叠式自编码机能识别面部特征、字母/数字字符或一些事物的泛化形式（比如一只鸟）。能学习输入数据中非常复杂的高层次抽象信息是堆叠式降噪自编码机的专长。

自编码机可以无限堆叠，而且事实证明，堆叠得越多，深度架构就越高效（此时主要约束变成了时间上的计算成本）。本章将堆叠 3 层自编码机来实践自然语言处理。

3.2.1　应用堆叠式降噪自编码机

介绍过作为深度学习架构的堆叠式降噪自编码机的优势和强大之处后，下面用真实数据集来实践所学内容。

这次不处理图像数据集，转而处理 OpinRank Review 数据集。该数据集来自加州大学欧文分校机器学习数据仓库，是一个包含 259 000 条 TripAdvisor 旅馆评价的文本数据集。该免费数据集包含评分（浮点数，取值范围为 1~5）和大量对旅馆的评价文本。下面用堆叠式降噪自编码机来处理该数据集，通过评价文本判断每个旅馆的评分。

> 下面将用自编码机分析预处理过的数据（可从本章的 GitHub 附件中获取），并探讨为下一章准备文本数据的技术。源数据的下载地址为 https://archive.ics.uci.edu/ml/datasets/OpinRank+Review+Dataset。

首先创建一个堆叠式降噪自编码机（以下称作 `SdA`）类。

```
class SdA(object):


    def __init__(
        self,
        numpy_rng,
        theano_rng=None,
        n_ins=280,
        hidden_layers_sizes=[500, 500],
        n_outs=5,
        corruption_levels=[0.1, 0.1]
    ):
```


前文提过，通过将一层自编码机的编码传送给下一层作为其输入项，构造出了 SdA。该类支持配置层数（体现在 hidden_layers_sizes 和 corruption_levels 向量的长度上，但不是由这两个变量设置的）和层大小（即不同的节点数，由 hidden_layers_sizes 设置）。如前所述，能配置连续的自编码机层是获得数据成功表示的关键。

然后构造变量来存储 SdA 的多层感知机(self.sigmoid_layers)和降噪自编码机(self.dA_layers ）元素。为了声明架构的深度，用 self.n_layers 参数声明所需要的 sigmoid 和降噪自编码机层的数量。

```
self.sigmoid_layers = []
self.dA_layers = []
self.params = []
self.n_layers = len(hidden_layers_sizes)

assertself.n_layers> 0
```

接着构造 sigmoid 和降噪自编码机层。首先根据输入向量的长度或前一层激活值的数量来确定隐藏层的大小，然后创建 sigmoid_layer 和 dA_layer，后者来自前面提到的 dA 类。

```
for i in xrange(self.n_layers):
    if i == 0:
        input_size = n_ins
    else:
        input_size = hidden_layers_sizes[i - 1]

if i == 0:
    layer_input = self.x
else:
    layer_input = self.sigmoid_layers[-1].output

sigmoid_layer = HiddenLayer(rng=numpy_rng,
                            input=layer_input,
                            n_in=input_size,
                            n_out=hidden_layers_sizes[i],
                            activation=T.nnet.sigmoid)

self.sigmoid_layers.append(sigmoid_layer)
```

```
self.params.extend(sigmoid_layer.params)

dA_layer = dA(numpy_rng=numpy_rng,
              theano_rng=theano_rng,
              input=layer_input,
              n_visible=input_size,
              n_hidden=hidden_layers_sizes[i],
              W=sigmoid_layer.W,
              bhid=sigmoid_layer.b)

self.dA_layers.append(dA_layer)
```

实现了堆叠式降噪自编码机的层后，最后还需要一个逻辑回归层来完成网络的多层感知机部分。

```
self.logLayer = LogisticRegression(
    input=self.sigmoid_layers[-1].output,
    n_in=hidden_layers_sizes[-1],
    n_out=n_outs
)

self.params.extend(self.logLayer.params)
self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)
self.errors = self.logLayer.errors(self.y)
```

至此，堆叠式降噪自编码机的架构就基本完成了。下一步需要生成 SdA 类要使用的训练函数。每个函数都会将每个小批次的索引（index）和其他一些元素作为参数，这里选择 corruption_level 和 learning_rate，这样可以在训练过程中调节它们（比如逐渐增大或减小）。另外，还定义了确定批次开始和结束的变量，分别为 batch_begin 和 batch_end。

> 动态调整学习率非常有用，其应用方式有两种。当技术开始收敛到合适的解决方案时，降低学习率将非常有用，否则网络可能在最优值附近上下摆动而无法达到最优。在很多情况下，将学习率与网络的性能测度挂钩也是一种有效的方法。如果误差率很高，最好做较大调整，直到误差率开始降低。

```
def pretraining_functions(self, train_set_x, batch_size):
    index = T.lscalar('index')
    corruption_level = T.scalar('corruption')
    learning_rate = T.scalar('lr')
    batch_begin = index * batch_size
    batch_end = batch_begin + batch_size

    pretrain_fns = []
    for dA in self.dA_layers:
        cost, updates = dA.get_cost_updates(corruption_level,
                                            learning_rate)
        fn = theano.function(
            inputs=[
                index,
```

```
                theano.Param(corruption_level, default=0.2),
                theano.Param(learning_rate, default=0.1)
            ],
            outputs=cost,
            updates=updates,
            givens={
                self.x: train_set_x[batch_begin: batch_end]
            }
        )
        pretrain_fns.append(fn)

    return pretrain_fns
```

所构造的预训练函数以每个小批次的 index 为输入，也可选用损坏水平或学习率。该函数执行预训练这一步骤，输出成本值和权重更新向量。

除了预训练外，还需要构造微调阶段的函数。在微调阶段，网络将在验证数据和测试数据间迭代，以优化网络参数。以下代码中的训练函数（train_fn）实现了一步简单的微调。valid_score 是一个 Python 函数，它用 SdA 在验证数据上生成的误差测度计算验证集得分。与之相似，test_score 在测试集上计算得分。

为了实现上述目标，首先需要建立训练集、验证集和测试集，每个阶段都需要两个数据集（集合 x 和集合 y），分别包含特征和类标签。验证和测试所需的小批次的数量是确定的，还需要构造一个索引（index）来追踪批次大小（并判断批次的起止位置）。每个批次都要进行训练、验证和测试，然后计算所有批次的 valid_score 和 test_score。

```
def build_finetune_functions(self, datasets, batch_size, learning_rate):

    (train_set_x, train_set_y) = datasets[0]
    (valid_set_x, valid_set_y) = datasets[1]
    (test_set_x, test_set_y) = datasets[2]

    n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
    n_valid_batches /= batch_size
    n_test_batches = test_set_x.get_value(borrow=True).shape[0]
    n_test_batches /= batch_size

    index = T.lscalar('index')


    gparams = T.grad(self.finetune_cost, self.params)


    updates = [
        (param, param - gparam * learning_rate)
        For param, gparam in zip(self.params, gparams)
    ]

train_fn = theano.function(
    inputs=[index],
```

```
    outputs=self.finetune_cost,
    updates=updates,
    givens={
        self.x: train_set_x[
            index * batch_size: (index + 1) * batch_size
        ],
        self.y: train_set_y[
            index * batch_size: (index + 1) * batch_size
        ]
    },
    name='train'
)

test_score_i = theano.function(
    [index],
    self.errors,
    givens={
            self.x: test_set_x[
            index * batch_size: (index + 1) * batch_size
        ],
            self.y: test_set_y[
            index * batch_size: (index + 1) * batch_size
        ]
    },
    name='test'
)

valid_score_i = theano.function(
    [index],
    self.errors,
    givens={
        self.x: valid_set_x[
            index * batch_size: (index + 1) * batch_size
        ],
        self.y: valid_set_y[
            index * batch_size: (index + 1) * batch_size
        ]
    },
    name='valid'
)


def valid_score():
    return [valid_score_i(i) for i inxrange(n_valid_batches)]

def test_score():
    return [test_score_i(i) for i inxrange(n_test_batches)]

return train_fn, valid_score, test_score
```

训练函数就绪后，启动堆叠式降噪自编码机，代码如下所示。

```
numpy_rng = numpy.random.RandomState(89677)
print '... building the model'
```

```
sda = SdA(
    numpy_rng=numpy_rng,
    n_ins=280,
    hidden_layers_sizes=[240, 170, 100],
    n_outs=5
)
```

请注意，此时应先尝试一种初始层数配置，看看效果如何。本例中的层数也是初始测试的产物。前面讲过，`SdA` 的训练有两个阶段：第一阶段是逐层预训练，遍历所有 `SdA` 层；第二阶段是基于验证集和训练集进行微调。

为了预训练 `SdA`，需要针对每层的训练设置损坏水平，并用之前定义的 `pretraining_fns` 来遍历各层。


```
print '... getting the pretraining functions'
pretraining_fns = sda.pretraining_functions(train_set_x=train_set_x,
                                            batch_size=batch_size)

print '... pre-training the model'
start_time = time.clock()
corruption_levels = [.1, .2, .2]
for i in xrange(sda.n_layers):

    for epoch in xrange(pretraining_epochs):
        c = []
        for batch_index in xrange(n_train_batches):
            c.append(pretraining_fns[i](index=batch_index,
                                        corruption=corruption_levels[i],
                     lr=pretrain_lr))

print 'Pre-training layer %i, epoch %d, cost ' % (i, epoch),

print numpy.mean(c)

end_time = time.clock()

print(('The pretraining code for file ' +
       os.path.split(__file__)[1] + ' ran for %.2fm' % ((end_time - start_time) / 60.)),
       file = sys.stderr)
```

现在可以初始化 `SdA` 类了，需要调用本书的 GitHub 库中的代码：MasteringMLWithPython/Chapter3/SdA.py。

3.2.2 评估堆叠式降噪自编码机的性能

堆叠式降噪自编码机的运行非常耗时。每层迭代 15 次，平均需要 11 分钟，因此在一个配置了 GPU 加速和单线程 GotoBLAS 的台式机上运行大约需要 500 分钟。

对于没有 GPU 加速的系统来说，网络的训练将耗时更长，因此建议在小得多的输入数据集

上运行另一种代码：MasteringMLWithPython/Chapter3/SdA_no_ blas.py。

结果很好，验证误差得分为 3.22%，测试误差得分为 3.14%。鉴于自然语言处理经常出现模棱两可或其他棘手问题，这个结果已经非常不错了。

显然，网络对 1 星和 5 星数据分类的正确率比中间水平的更高，因为缺少极端倾向或情绪化语言所导致的语义模糊。

输入数据可分类的部分原因在于有效的特征工程。虽然特征工程很耗时，有时还会出错，但正常运行的特征工程和最优模型相结合就能达到很高的准确率。第 6 章将应用一些技巧来对数据进行预处理。

3.3 扩展阅读

谷歌大脑团队的 Quoc V. Le 撰写了一篇自编码机（还涉及其他话题）的概述，参见 https://cs.stanford.edu/~quocle/tutorial2.pdf。

本章使用的 Theano 文档参见 http://deeplearning.net/tutorial/contents.html，该文档是本章内容的基础，因为 Theano 是本章使用的主要软件库。

3.4 小结

本章介绍了自编码机，这种降维技术非常高效，有许多独特的应用；还介绍了堆叠式降噪自编码机背后的理论。堆叠式降噪自编码机是对自编码机的拓展，即通过深度架构将一系列自编码机堆叠起来。堆叠式降噪自编码机可用于解决棘手的自然语言处理问题，并且效果极佳，这在旅馆评价的情感分析问题中达到了较高的准确度。

下一章将讨论监督深度学习方法，包括**卷积神经网络**。

第 4 章

卷积神经网络

本章介绍如何应用卷积神经网络（convolutional neural network，CNN），这可能是最知名的深度架构了。本章主要内容如下：

- CNN 的拓扑结构和学习过程，包括卷积层和池化层
- 用 CNN 组件构建网络架构
- 用 Python 代码实现 CNN 架构来解决著名的图像分类问题

4.1 CNN 介绍

在机器学习领域，人们长期致力于使用代码构造类似于生物结构的架构，例如多层感知机神经网络，其拓扑结构和学习过程都受到了人脑神经元的启发。

实践证明，这种（仿生的）偏好确有成效。针对特定任务，我们能以最优的特定生物学结构为模板和线索，设计和创建出强大的学习模型。

CNN 的灵感来自视觉皮层，后者是大脑中处理视觉输入的区域。视觉皮层的功能多样，因此可以有效地处理视觉数据，而且它包含许多受体细胞，因而能检测到视野重叠区中的光。所有受体细胞都执行同一种卷积运算，即它们处理输入的方式相同。CNN 的设计包含了这些功能，因此其拓扑结构与其他神经网络有着显著差异。

毫不夸张地说，目前 CNN 是人工智能和机器学习中许多有影响力的先进技术的基础。CNN 的各种变体广泛应用于复杂的视觉和语言等领域，示例如下。

- 谷歌开发了一系列专业 CNN 架构，包括 GoogLeNet，它有 22 层。此外，谷歌的 DeepDream 项目（因训练过度、生成的图像风格梦幻而得名）也使用了 CNN。
- 卷积网络已经训练得能下围棋（对人工智能来说，围棋一直是挑战），与高排名玩家对战的胜率在 85%~91%。
- Facebook 的人脸识别（DeepFace）也使用了卷积网络。
- 许多团队都用 CNN 开发新一代智能应用，例如百度、微软研究院、IBM 和 Twitter。

近年来，目标识别大赛（比如 2014 年的 ImageNet 大赛）的胜者大都应用了极为专业的 CNN，或 CNN 与其他架构相结合的多模型集成。

第 8 章将介绍如何构造并有效应用集成方法，本章则关注 CNN 在大规模图像分类问题上的成功应用。

4.1.1　CNN拓扑结构

有的读者可能熟悉 CNN 的架构。这是一个非循环图，层内节点数逐渐减少，每一层都反馈到下一层。这与许多知名网络拓扑结构相似，比如多层感知机。

与其他大多数网络最显著的区别是，CNN 的所有神经元都相同。所有神经元包含相同的参数和权重值。显然，这直接减少了网络控制的参数个数，大大提升了存储效率。此外，由于需要控制和计算的自由参数更少，因此往往能提高网络的学习速度。稍后会介绍，共享权重也使得 CNN 能学习输入（比如输入图像或音频信号）中与位置无关的特征。

卷积网络与其他架构的另一个关键区别是，节点之间的连接性受限，因此能构造局部空间连接模式，即反馈给特定节点的输入将限制在邻近的感受野内的节点上。这可以是空间上连续的，比如图像数据。每个神经元的输入最终都取自图像的连续子集。对音频信号来说，输入可能是连续的随时间变化的滑动窗口。

下面举例说明，并讲解卷积网络如何通过特定节点处理图像的特定部分。CNN 第一层的节点将会分配到输入图像的子集。在本例中，假设每个节点都得到 3 像素×3 像素的图像子集。这些子集能构成整个图像，节点的输入不存在重叠，也没有间隔。（注意，以上这些条件对 CNN 来说并不一定正确。）每个节点分配到 3 像素×3 像素的图像子集（节点的感受野，也称“窗口”），并输出转换后的输入。这里先不考虑该转换的具体情况。

通常输出项会传送至第二层的节点。本例假设第二层采用第一层节点的所有输出的子集。比如采用原始图像的连续 6 像素×6 像素子集，即其感受野包含上一层 4 个节点的输出，见图 4-1。

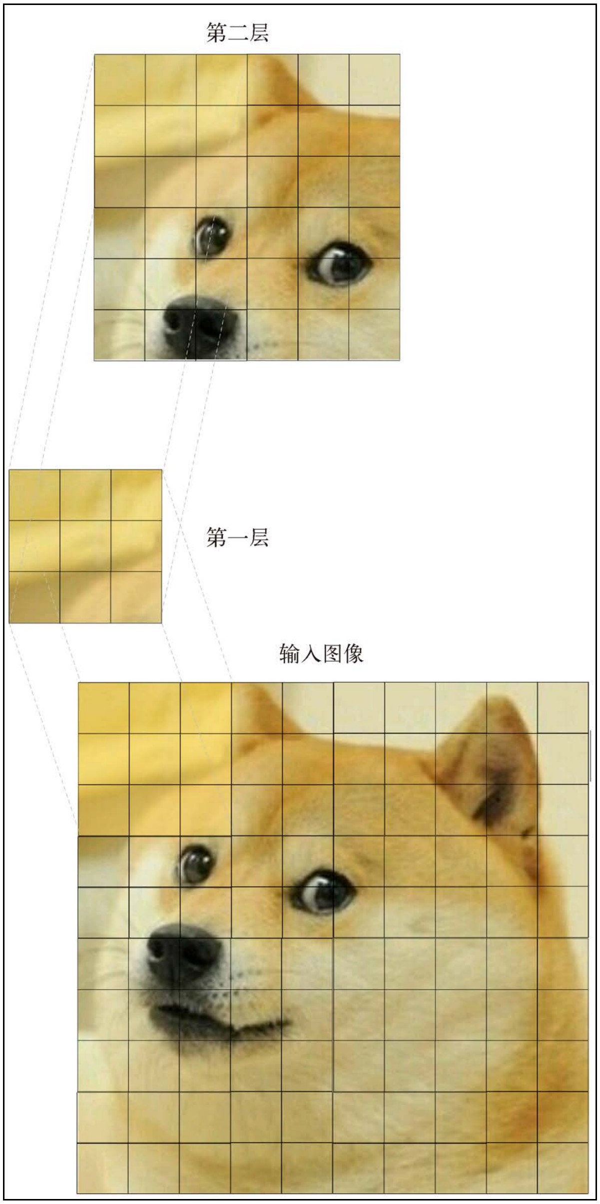


图　4-1

每一层都是**可组合的**，一个卷积层的输出可以作为下一层的输入。这和第 3 章提到的作用相同，连续层能提取更高层次、更抽象的特征。此外，在向下构造架构（即添加新层）时，数据的表示将与更大范围内的像素空间相关。通过堆叠层，最终可以生成整个输入项的全局表示。

1. 卷积层

前面讲过，为了防止每个节点学习无法预测（而且很难调节）的一系列局部自由参数，必须共享整个层的权重。为了确保完全精确，应用于卷积层的滤波器是一组滤波器，其滑动作用（即卷积）于整个输入数据集，从而形成输入数据的二维激活映射，称为“特征映射”。

滤波器由 4 个超参数控制：大小、深度、步长（也称“步幅”）和补零（也称“0 填充”）。滤波器的大小就是其范围（显然，等于高乘宽，滤波器不一定是矩形）。更大的滤波器会增大重叠面积，也会提升分类准确度，稍后详述。关键是，增大滤波器会使得输出项更大。控制卷积层输出的尺寸将在很大程度上影响网络的效率。

深度指的是连接输入项同一区域的层内节点数。理解此概念的关键在于认识到，无论人还是网络，观察图像时都会处理很多性质[①]。参见PS中的图像调整滑块。就其本身而言，深度有时也称作“维度”，它与图像的复杂程度密切相关。这不是指图像内容的复杂程度，而是指准确描述图像所需的通道数。

将节点映射为输入图像的 RGB 值后，合理的深度设置可以生成描述颜色的通道，因此深度通常设置为 3（尤其是在第一个卷积层）。请注意，有些节点会学习表达输入图像不易描述的特性，从而使得 CNN 对图像的学习更精准。增大深度超参数使得节点可以学习更抽象的输入信息的编码，这样做有利也有弊。

过小的深度参数会导致结果较差，因为网络的表达深度（从通道数的角度看）达不到要求，无法准确描述输入数据的特征。该问题与特征不足的问题相似，但前者更容易解决。可以上调网络的深度来提升 CNN 的表达深度。

过小的深度参数不但多余，甚至会影响之后的性能。不妨考虑在网络配置阶段通过超参数优化来选择合适的深度值，比如肘部法或其他技术。

步长度量的是神经元之间的空间。步长为 1 将使得输入中的每个元素（比如图像输入的每个像素）都成为滤波器实例的中心，从而带来高度重叠和极其庞大的输出。增大步长可以减少感受野中的重叠，缩减输出的大小。微调 CNN 的步长就是在准确率和输出大小间做权衡。通常较小的步长效果更好。另外，步长为 1 可以控制池化层的向下采样和适度缩减。

图 4-2 生动阐释了**深度**和**步长**。

① 不同的图像性质对应模型不同的滤波器。——译者注

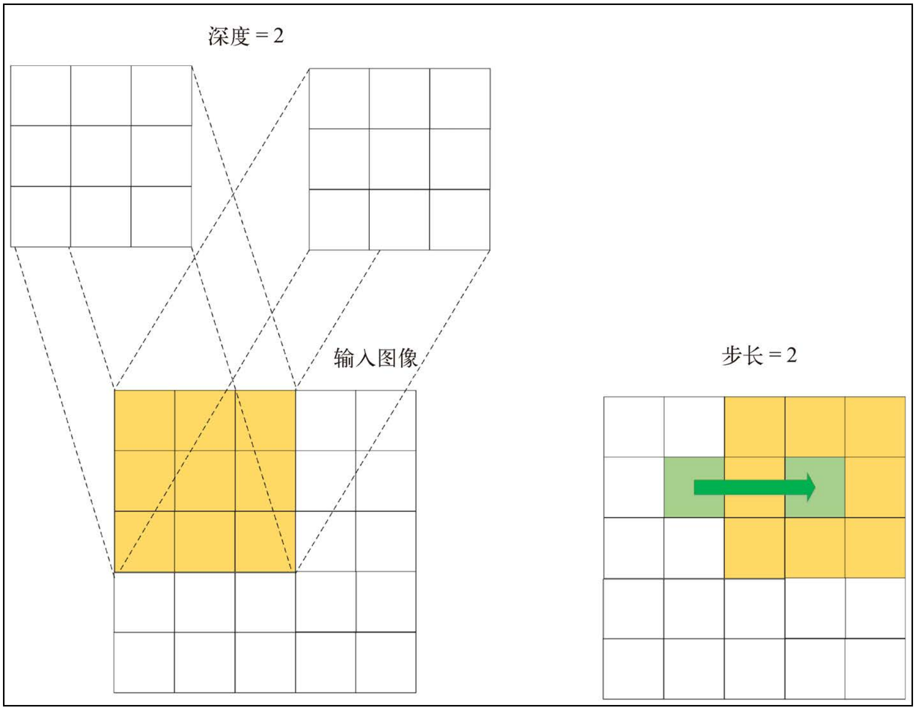


图　4-2

最后一个超参数**补零**则提供了一种便利。补零指设置每个感受野的边缘值（边界），这可以降低该层的输出大小。可以将感受野最外层或最外几层的像素值设为 0，输出大小也会相应改变。这种做法存在限制，如果设置完补零和步长后，输入数据的某些区域和滤波器不相接，显然有问题。更普遍的情况是，加重补零程度会降低效率，因为过于粗糙的编码会导致学习特征的难度加大（稍后详述）。

但补零仍是有用的工具，可以通过补零将输入项和输入项的大小调为一致。这种做法很常用。通过补零来确保输入层和输出层相等，便于控制步长和深度值。如果没有补零这一步骤，可能需要做大量工作来追踪输入大小、控制网络参数，只为确保网络可以正常运作。另外，补零也能提升性能，如果不这么做，CNN 在滤波器边缘的性能就会逐渐降低。

为了在定义 CNN 时调整节点数、最适宜步长和连续层的填充值（也称“填充像素”），必须知道上一层的输出大小。可以将上一层的输出项（O）作为输入图像大小（W）、滤波器大小（F）、步长（S）和补零数量（P）的函数，如下所示。

$$O = \frac{W - F + 2P}{S + 1}$$

如果 O 不是整数，滤波器就不会完美贴合输入项，而会越过输入项的边缘。这会给训练带来一些问题（比如报错）。通过调整步长值，可以找到 O 的一个整数解，然后有效地进行训练。在给定其他超参数值和输入大小的情况下，通常会将步长限制在可能的范围内。

前面讨论了如何通过超参数正确配置卷积层，但未讨论卷积过程本身。与加法和导数相同，卷积也是一种数学运算，在信号处理领域应用颇多，也常用于简化复杂公式。

大致说来，卷积是对两个函数的运算，其结果是生成第三个函数，该函数是两个原始输入函数其中一个的修改版本。对 CNN 中的卷积而言，两个原始输入函数中的第一个就是网络的输入。如果应用于图像，那么卷积要应用于二维空间（图像的宽和高）。输入图像通常是 3 个像素矩阵，分别为红色、蓝色、绿色通道的矩阵，每个通道的值的范围是 0~255。

> 这里需要引入**张量**（tensor）的概念。在深度学习中，张量经常指代输入数据的 n 维数组或矩阵，它与矩阵或数组非常相似。本章和第 9 章将详细介绍张量（第 9 章将介绍 TensorFlow 库）。值得注意的是，该术语在机器学习社区重获关注，很大程度上归功于谷歌机器智能研究团队。

卷积运算的第二个成分是卷积核，这是一个以浮点型数值为元素的矩阵，用作输入矩阵的滤波器。卷积运算的输出项是特征映射。卷积运算的原理是，在输入上滑动滤波器，在每个实例上计算两个参数的点积，然后记录在特征映射上。如果卷积层的步长为 1，就需要对输入图像的每个像素进行运算。

卷积的主要优势是能降低对特征工程的需求。构造、控制复杂的核和进行高度专业化的特征工程是项艰巨的工作，并且在某个领域表现优异的特征工程往往不适合其他领域，这就更为棘手了。第 7 章将详细介绍特征工程，而卷积网络是行之有效的候选方案。

CNN 能逐渐提升其核对输入数据筛选信息的能力，从而自动优化核。多核并行学习加速了该过程。这就是前面提过的特征学习。这种学习既节省了时间，也有助于解决很多问题。与之前讲过的堆叠式降噪自动编码机和深度信念网络实现类似，稍后将学到的特征传递到一个更简单、更浅的神经网络，并用这些特征对输入图像进行分类。

2. 池化层

堆叠卷积层能构造拓扑结构，从而为复杂的含噪输入数据高效构建特征，以形成特征映射。但卷积层并不是深度网络的唯一组件。卷积层通常会和池化层交替出现。池化是对特征映射进行的运算，它将多个特征值聚合成一个值，其途径多为**最大值**（最大池化，max-pooling）、**均值**（平均池化，mean-pooling）或**加和**（加和池化，sum-pooling）运算。

池化这种方法非常自然，优势显著。如果不对特征映射进行聚类，那么特征可能会过多。下面即将进行分类的 CIFAR-10 数据集包含 60 000 幅 32 像素×32 像素的图像，假设对每幅图像的 8

像素×8 像素子集学习 200 个特征，则最后得到的输出向量的大小为(32−8+1)×(32−8+1)×200，即对每幅图像学习了 125 000 个特征。卷积会生成大量特征，其计算成本高昂，同时也会引发严重的过度拟合问题。

池化运算的另一个主要优势是较为稳健，能够应对高维含噪数据中的许多小偏差和变化。具体而言，池化能防止网络学习过于个别的特征位置（即过度拟合），这显然是图像处理和识别问题的必要条件。通过池化，网络不再将输入数据的抽取特征固定在特定位置，从而提升了泛化能力，这叫作**平移不变性**（translation-invariance）。

最大池化是最常用的池化运算，因为它专注于问题中最具响应性的特征，理论上可以为图像识别和分类找出最佳特征。类似地，最小池化适用于需要进行额外工作来防止分类过于敏感或过度拟合的情况。

显然，通过直接池化方法（比如最大池化）快速部署模型是明智的。然而，为了确保模型能在后续迭代中持续提升网络性能，需要判断池化运算有没有改进的可能。可以自定义池化运算，这没有什么实际限制。找到更有效的二次抽样方法或其他聚类方法可以大幅提升模型性能。

用 `theano` 代码实现最大池化非常简单，如下所示。

```
from theano.tensor.signal import downsample

input = T.dtensor4('input')
maxpool_shape = (2, 2)
pool_out = downsample.max_pool_2d(input, maxpool_shape, ignore_border=True)
f = theano.function([input],pool_out)
```

`max_pool_2d` 函数将 n 维 `tensor` 和降尺度因子（本例中指 `input` 和 `maxpool_shape`）作为参数，其中 `maxpool_shape` 是一个长度为 `2` 的元组，包含输入图像的宽度和高度降尺度因子。然后 `max_pool_2d` 会根据元组向量的两个维度进行最大池化运算。

```
invals = numpy.random.RandomState(1).rand(3, 2, 5, 5)

pool_out = downsample.max_pool_2d(input, maxpool_shape, ignore_border=False)
f = theano.function([input],pool_out)
```

`ignore_border` 判断要考虑或舍弃边缘值。假设 `ignore_border = True`，那么这次最大池化运算将生成如下矩阵。

```
[[ 0.72032449  0.39676747]
[ 0.6852195   0.87811744]]
```

如上所示，池化是一种简单运算，而结果令人满意（本例中 5×5 的输入矩阵简化成了 2×2）。然而，这种运算方式也有劣势。Geoffrey Hinton 对此做出了相当有趣的点评。

“在 CNN 中进行池化运算是严重的错误，它运行得那么好简直是灾难。

如果池之间不存在重叠，池化就会丢失重要的位置信息，而我们需要用这些信息来检测物体组成部分间的准确关系。的确，如果池的重叠部分足够多，就会将特征的位置‘粗编码’以准确存储（我在 1986 年发表的有关‘分布式表达’的文章中解释了这种效应），但我现在不认为粗编码是表示物体相对于观察者的姿态（指位置、朝向和大小）的最佳方法了。”

语出惊人，但不无道理。Hinton 认为，作为聚合方法，池化和其他聚合方法的效果完全相同，即把数据降维到更简单、信息更少的形式。这无伤大雅，但 Hinton 考虑得更长远。

即使将每个池中的数据都简化为一个值，仍可寄希望于空间中重叠的几个池还能保留特征编码。（这就是 Hinton 所说的“粗编码”。）这也是个非常直观的概念。假设听到一段有噪声的无线电频段信号，即使只辨别出 1/3 的单词，也很可能听出船舶发出的遇险信号！

Hinton 又指出，粗编码不是学习姿态（位置、朝向和大小）的有效手段。根据视角的不同，物体的排列方式多样，这使得两幅图像很难完全相同，而姿态的多样性对使用池化的卷积网络而言是一个巨大挑战。这意味着，只有攻克这一挑战才能突破图像分类的上限。

然而，目前普遍认为即使承认以上所有论断，CNN 中的池化运算仍在效率和平移不变性上有着巨大优势。这是目前最好的选择了。

Hinton 还给出了 CNN 的一个替代方案——**转化自编码机**。转化自编码机能为精度要求高的学习任务（比如人脸识别）提升精度，而池化运算会降低精度。如果对转化自编码机感兴趣，参见 4.2 节，其中提供了一些阅读材料。

前面详细介绍了 CNN，包括其组件、运行原理和超参数。将理论付诸实践前，有必要介绍一下如何用这些组件构建工作架构。下面首先讲解如何训练 CNN。

3. 训练 CNN

CNN 的训练方法与前几章讲的训练方法类似。卷积架构本身可用于预训练更简单的网络架构（比如多层感知机）。反向传播算法是预训练阶段计算梯度的标准方法。在此过程中，每层都会经历以下 3 个步骤。

- **正推法**（forward pass）：将所有特征映射与对应核的权重进行卷积并求和，以计算出下一层的每个特征映射。
- **逆推法**（backward pass）：将转置的内核权重与输出对应的梯度进行卷积，以计算输入对应的梯度。
- 计算每个核的损失，从而按需调整每个核的权重。

重复该过程能提升核的性能，最终实现收敛。这里需要得到一系列特征，从而让受限网络高效地对这些特征进行分类。

该过程可能会执行得很缓慢，即使是在非常先进的 GPU 上运行。一些新技术能加速该训练过程，例如用**快速傅里叶变换**来加速卷积过程（卷积核大小与输入图像几乎相等时）。

4. 汇总组件

前面探讨了构建 CNN 所需要的元素，下面介绍如何组合这些元素以构造有效的卷积网络，以及哪种组合方式更有效。下面以一些现成的 CNN 架构为指导，讲解模型的常规用法和适用领域。

最知名的卷积网络可能就是 Yann LeCun 的 LeNet 了。自 20 世纪 80 年代末 LeNet-1 问世以来，该网络经历了多次迭代，在处理手写数字和对图像进行分类等方面的效果越来越好。LeNet 是用卷积层和池化层交替堆叠，并以多层感知机收尾而构成的，如图 4-3 所示。

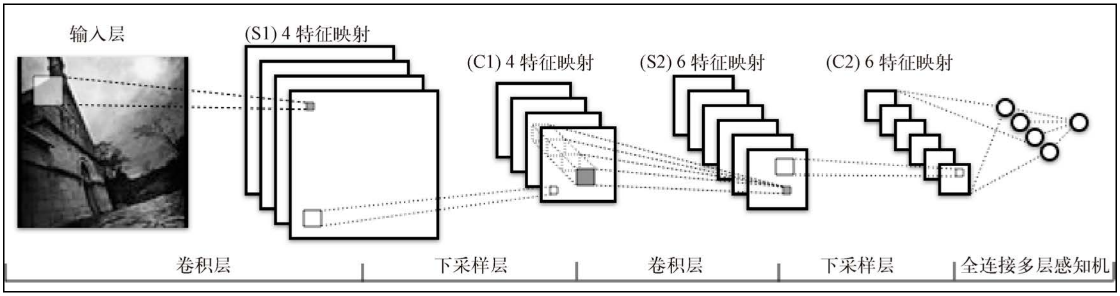


图 4-3

如前所述，每一层都和其他层存在部分连接，而多层感知机是全连接层。每一层都会部署多个特征映射（通道），因此可以构造更复杂的滤波器集。稍后将看到，在一层中使用多个通道是在许多前沿案例中采用的一种强大的技术。

常用最大池化层对输出项降维，以便其与输入项相匹配并控制输出规模。如何实现池化，特别是如何设定卷积层和池化层的相对位置，是结果出现差异的关键原因。通常将一系列运算视作一层，并将输出传递给一个全连接层，如图 4-4 所示。

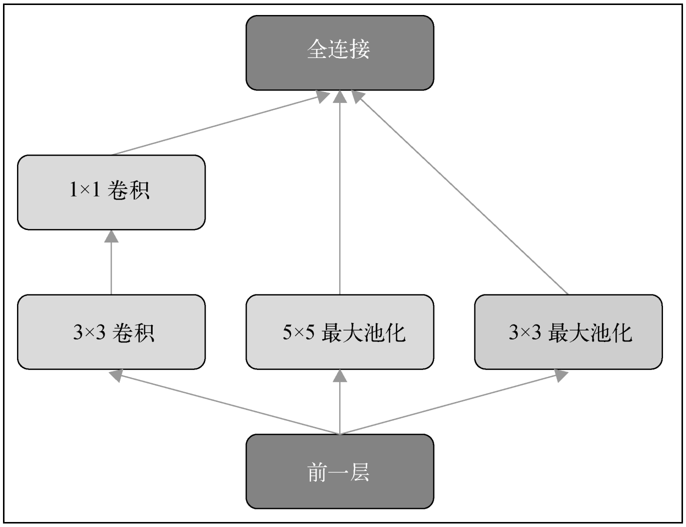


图　4-4

虽然该网络架构无法解决实际问题，但有助于说明用组件构建网络存在多种方式。网络的构建方法、架构的复杂程度，都应与网络要解决的问题有关。不同问题的解决方法可能天差地别。

在接下来要研究的 LeNet 实现案例中，每层都包含多个卷积层，每个卷积层后跟着一个最大池化层，如图 4-5 所示。

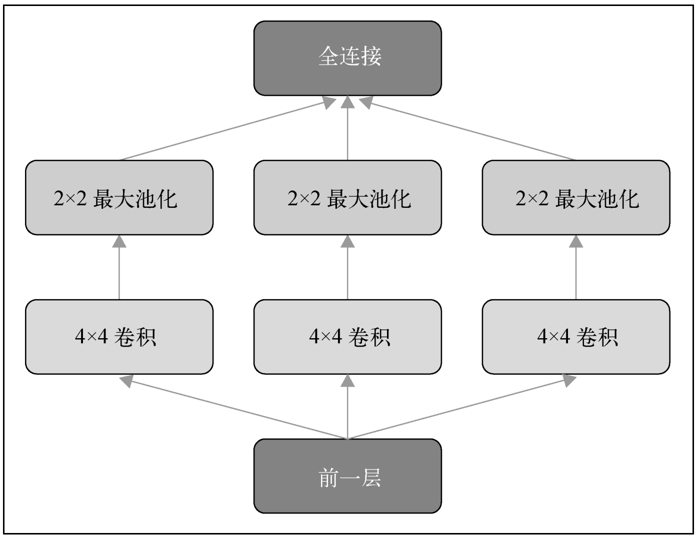


图　4-5

可以用该架构简单、快速地浏览一些初始用例，但整体表现并不像后面要讲到的先进应用那么理想。因此，人们设计了许多应用广泛的学习架构，以解决各种棘手的问题，其拓扑结构值得研究。谷歌的 Inception 网络是最知名的卷积架构之一，现在常称作 GoogLeNet。

GoogLeNet 的设计初衷是为了解决计算机视觉问题，包括网络图像数据，即现实中拍摄的图片，其姿态、灯光、遮挡和杂波存在巨大差异。GoogLeNet 在 2014 年的 ImageNet 大赛中大放异彩，在测试数据集上的误差率只有 6.7%。ImageNet 图像尺寸小、内容差异大，涉及种类多。这些种类可能看起来非常类似（比如树的不同种类），而网络架构必须成功辨别它们，例如图 14-6 所示的 ImageNet 图像。



图 4-6

针对该需求，GoogLeNet 架构在 ImageNet 中取得了优异成绩，并在许多关键方面超越了 LeNet 模型。GoogLeNet 基本层的设计称作 Inception 模块，由以下几部分组成（见图 4-7）。

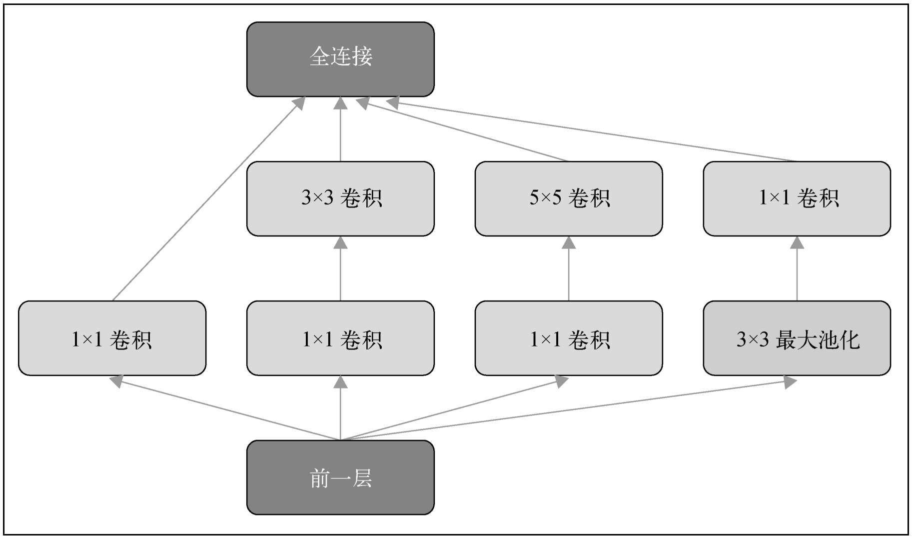


图 4-7

其中 1×1 的卷积层后跟着**修正线性单元**（rectified linear units，ReLU）。该方法常用于语音建模和音频建模，因为 ReLU 可以高效地训练深度模型，无须预训练，也没有其他激活函数存在的梯度消失问题。关于 ReLU 的更多信息，参见 4.2 节。DepthConcat 元素提供了连接函数，可以巩固多个单元的输出，并大大节约训练时间。

GoogLeNet 将这样的层连接起来，构造出完整的网络。Inception 模块在 GoogLeNet 网络中多处（9 处）存在。这也表明，网中网（network in network，NIN，由链接网络模块构成的深度架构）方法将一直是深度学习领域的有力竞争者。4.2 节提到的一篇文章介绍了 GoogLeNet 并阐述了 Inception 模块是如何集成到网络中的。

除了采用 Inception 模块的堆叠技巧外，GoogLeNet 还有其他可称赞的地方。前几层往往比较简单，首先使用单通道卷积层和最大池化层。另外，GoogLeNet 还在多处引入了主体架构上的分支——平均池化层，其输出项传递至附加的 softmax 分类器。这些分类器旨在改善传递回网络内低层的梯度信号，提高网络上层和中层的性能。GoogLeNet 并没有在网络的最后一层执行大规模但可能较模糊的反向传播，而是有一些中间更新源。

该实现表明，GoogLeNet 和其他顶级 CNN 之所以成功，主要是因为它们使用了本章所讲的高可用组件并进行了有效的配置。介绍过了卷积网络的组件，以及它们如何构成先进的网络，下面用这些技术解决具体问题。

4.1.2　应用CNN

下面用 CNN 处理图像数据。前几章处理过的图像数据（包括 MNIST 数字数据集）就是非常有用的训练数据集（包含了许多有价值的现实应用，比如自动审阅）。然而这些数据与其他所有图像数据或视频数据不同的一点是：大部分可视数据都含有大量噪声。

问题变量包括姿态、灯光、遮挡和杂波，它们可以独立表达，也可以以多种形式结合起来表达。这意味着，不受数据噪声影响的函数很难构造，而且这样的函数通常是复杂的非线性函数。第 7 章将探讨“白化”等技术如何应对这些挑战，但仅依靠这些技术无法得到理想的分类结果（至少在没有投入大量时间的情况下）。截至目前，处理图像数据噪声的最有效方法是深而不广的架构（高维层很少的神经网络极易受到过度拟合和泛化问题的影响），这在许多情景中出现过了。

至此可归纳出深度架构的原理：深度架构的连续层会使用上一层的推理和计算过程。因此，深度架构可以构建一种数据表示，这种表示无须在任何单独的层上进行大量反复计算，就能改善连续层。这能在相对较短的时间内让模型对大规模含噪图像数据分类达到较高的准确度，而无须进行大量特征工程。

介绍过对图像数据建模的挑战以及深度架构的优势后，下面对一个实际分类问题应用 CNN。

如前所示，首先通过一个小例子进一步说明深度网络架构。本章以一个经典的图像处理问题（CIFAR-10）为例，该数据集包含 60 000 幅 32 像素×32 像素的彩色图像。这些图像可分为 10 类，每类包含 6000 幅图像。数据已分为 5 个训练集和 1 个测试集。每个子集的类别和对应的一些图像如图 4-8 所示。

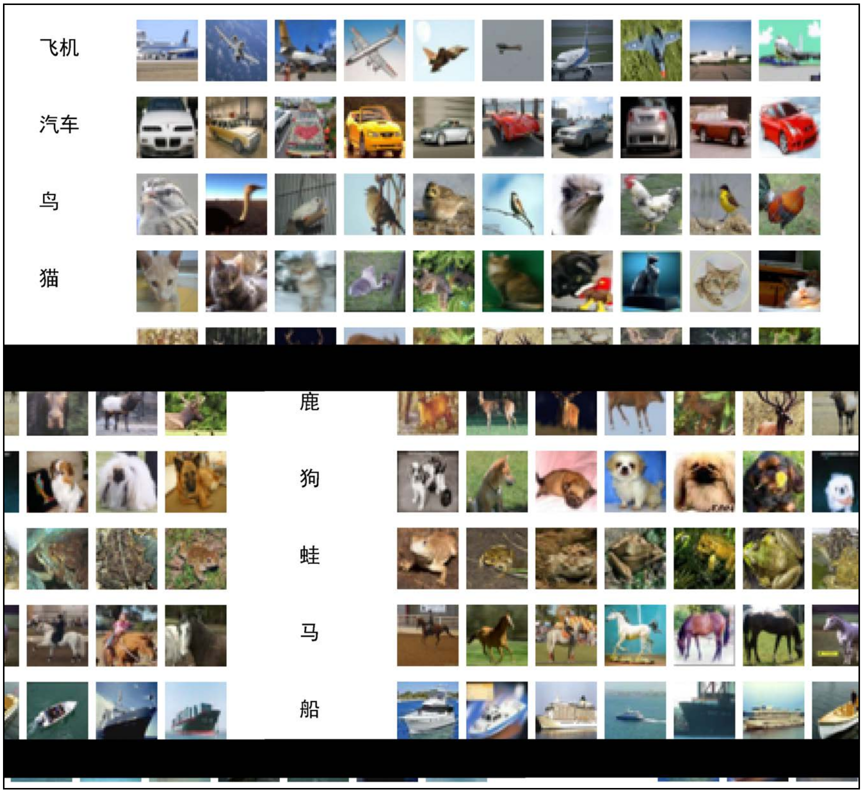


图　4-8

尽管从某种程度上讲，业内已经开始处理其他数据集（比如 ImageNet）了，但 CIFAR-10 一直是图像分类难以跨越的障碍。许多数据科学家尝试构造架构对该数据集进行分类，以期精度达到人类水平（人类对此的误差率约为 6%）。

2014 年 11 月，Kaggle 举办了一场比赛，目标是尽可能准确地对 CIFAR-10 进行分类。获胜者的分类精度达到了 95.55%，他们使用的正是卷积网络和网中网。第 8 章将讨论如何对该数据集进行分类，以及可用的先进技术。下面首先尝试用卷积网络来进行分类。

- 对图像应用滤波器，并查看结果。
- 查看 CNN 创建的权重。
- 探究网络有效和无效的区别。

本章将采用一种新方法，其重要性一用便知。如前所述，为解决不同问题而开发的深层网络架构在结构上会有所不同。

重要的是针对特定问题构造网络架构，以便调整网络来适应一系列实际问题。为此，要将网络组件模块化，便于以各种方式重组组件来快速构建网络。前面介绍了模块化的影响，如何将其应用于网络值得探讨。

前面讲过，卷积网络对庞大且多样、包含成千上万幅图像的数据集的分类功能尤其强大。因此，不妨尝试用 CNN 对 CIFAR-10 进行分类。

在建立卷积网络的过程中，首先要定义一个可用类，并初始化相关网络参数（尤其是权重和偏差）。前面讲过这种方法。

```
class LeNetConvPoolLayer(object):

    def __init__(self, rng, input, filter_shape, image_shape,
                 poolsize=(2, 2)):

        assert image_shape[1] == filter_shape[1]
        self.input = input

        fan_in = numpy.prod(filter_shape[1:])
        fan_out = (filter_shape[0] * numpy.prod(filter_shape[2:]) numpy.prod(poolsize))

        W_bound = numpy.sqrt(6. / (fan_in + fan_out))
        self.W = theano.shared(
            numpy.asarray(
                rng.uniform(low=-W_bound, high=W_bound,
                            size=filter_shape),
                dtype=theano.config.floatX
            ),
            borrow=True
    )
```

创建偏差变量前，先查看一下已有元素。`LeNetConvPoolLayer` 类用于按照 LeNet 层结构实现完整的卷积层和池化层。该类包含多个初始参数。

前面介绍过 `rng` 参数了，它可以将权重初始化为随机值。前面还提到了 `input` 参数。在大多数情况下，图像输入多采用图像符号张量（symbolic image tensor）的形式。图像输入大小由 `image_shape` 参数设定，这是一个元组或长度为 4 的列表，用于描述输入的维度。在经过连续层时，该参数会逐渐减小。作为元组时，该参数只声明输入的高度和宽度；作为长度为 4 的列表时，参数的含义依次如下。

- 子集大小
- 输入特征映射的数量
- 输入图像的高度
- 输入图像的宽度

image_shape 能声明输入的大小，filter_shape 能声明滤波器的维度。当后者是长度为 4 的列表时，其元素的含义依次如下。

- 应用的滤波器（通道）数量
- 输入特征映射的数量
- 滤波器的高度
- 滤波器的宽度

高度和宽度的输入可以不需要任何额外参数。最后一个 poolsize 参数描述的是缩小规模因子，表现为长度为 2 的列表，第一个元素代表行数，第二个元素代表列数。

定义好这些值后，就可以用它们定义 LeNetConvPoolLayer 类了。定义 fan_in 时，将每个隐藏单元的输入项设为输入特征映射数量的倍数——滤波器的高度和宽度。类似地，可以简单定义 fan_out，这是一个梯度值，基于输出特征映射的倍数（即特征的高度和宽度除以池化规模的值）计算所得。

4

然后将偏差定义为一组一维张量，每个偏差对应一个输出特征映射。

```
b_values = numpy.zeros((filter_shape[0],),
                        dtype=theano.config.floatX)
self.b = theano.shared(value=b_values, borrow=True)

conv_out = conv.conv2d(
    input=input,
    filters=self.W,
    filter_shape=filter_shape,
    image_shape=image_shape
)
```

调用该函数就能定义一个使用先前定义的滤波器的卷积运算。有时为了高效应用一个函数，需要掌握许多理论知识。下面用 max_pool_2d 构建一个相似的池化运算。

```
pooled_out = downsample.max_pool_2d(
    input=conv_out,
    ds=poolsize,
    ignore_border=True
)


self.output = T.tanh(pooled_out + self.b.dimshuffle('x', 0, 'x', 'x'))

self.params = [self.W, self.b]

self.input = input
```

接着将偏项重塑成大小为(1, n_filters, 1, 1)的张量，然后添加偏置。这样可以使偏置影响每个特征映射和小批次。至此构建基础 CNN 的所有组件就准备好了，下面构建 CNN。

```
x = T.matrix('x')
y = T.ivector('y')
```

该过程非常简单，按顺序构建层，将参数传至先前声明的类。首先构建第一层。

```
layer0_input = x.reshape((batch_size, 1, 32, 32))

layer0 = LeNetConvPoolLayer(
    rng,
    input=layer0_input,
    image_shape=(batch_size, 1, 32, 32),
    filter_shape=(nkerns[0], 1, 5, 5),
    poolsize=(2, 2)
)
```

首先重塑输入，并传入所有选定的小批次。CIFAR-10 图像的维度为 32×32，即高度和宽度的输入大小也是 32×32。过滤过程会将该输入的大小在每个维度上减至 28（32–5+1）。池化则将该输入在每个维度上减半，从而构建一个大小为`(batch_size, nkerns[0], 14, 14)`的输出层。

这是完整的第一层。下面用相同的代码构建第二层。

```
layer1 = LeNetConvPoolLayer(
    rng,
    input=layer0.output,
    image_shape=(batch_size, nkerns[0], 14, 14),
    filter_shape=(nkerns[1], nkerns[0], 5, 5),
    poolsize=(2, 2)
)
```

参照上一层，这一层的输出尺寸为`(batch_size, nkerns[1], 5, 5)`。至此一切顺利。下面将该输出传至下一个全连接 sigmoid 层。首先需要将输入维度平铺成二维，根据目前传至该网络的值，这一层的输入应是形状为`(500, 1250)`的矩阵。因此设置一个恰当的 `layer2`。

```
layer2_input = layer1.output.flatten(2)

layer2 = HiddenLayer(
    rng,
    input=layer2_input,
    n_in=nkerns[1] * 5 * 5
    n_out=500,
    activation=T.tanh
)
```

添加最后的逻辑回归层，并计算全连接 sigmoid 层的值，进一步完善了网络。

执行以下代码。

```
    x = T.matrix(CIFAR-10_train)
    y = T.ivector(CIFAR-10_test)

Chapter_4/convolutional_mlp.py
```

结果如下所示。

```
Optimization complete.

Best validation score of 0.885725 % obtained at iteration 17400, with
test performance 0.902508 %

The code for file convolutional_mlp.py ran for 26.50m
```

验证精度相当不错。但未达到人类水平（94%），同时也不是 CNN 所能取得的最好成绩。

例如 4.2 节要讲的基于 Torch 并结合 dropout 技术（第 3 章介绍过）实现的 CNN，以及用于降低训练过程中的协变量漂移的**批归一化**（batch normalization）技术，验证精度可达 92.45%。

但 88.57%的验证精度与之没什么两样，该实践相当于初步尝试用高效网络架构解决 CIFAR-10 问题。更重要的是，本章教授了很多有关如何高效配置和训练 CNN 的知识。

4.2 扩展阅读

近来，人们对卷积网络的兴趣愈发高涨，相关资料也日益丰富。Andrej Karpathy 课程的课堂笔记值得阅读：http://cs231n.github.io/convolutional-networks/。

如果对某个最佳实现的细节感兴趣，可继续探究如下网络。

- 谷歌的 GoogLeNet。
- 谷歌 Deepmind 的 Go-playing 项目。
- AlphaGo。
- Facebook 的人脸识别网络架构 DeepFace。
- ImageNet LSVRC-2010 大赛获胜作品（http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf），由 Krizhevsky、Sutskever 和 Hinton 开发。
- Sergey Zagoruyko 在 Torch 上用批归一化实现的卷积网络（http://torch.ch/blog/2015/07/30/cifar.html）。

4.3 小结

本章内容丰富。首先介绍了一种新的神经网络——CNN，讲解了其背后的理论和常用的网络架构，探讨了由谷歌和百度等企业提出的一些先进的网络设计原则，还介绍了拓扑结构以及网络的运行原理。

然后构建了 CNN，并将该网络应用于 CIFAR-10 数据集。我们用模块化的 CNN 代码构建了功能架构，并在对 10 类图像数据的分类中达到了理想精度。虽然精度仍与人类水平存在差距，但该差距已在慢慢缩小！第 8 章将继续探讨本章所讲的内容，并在更高层次应用这些技术。

第 5 章

半监督学习

5.1 简介

前几章用先进技术解决了一系列数据问题。每个案例都应用技术操作数据集并取得了不错的成绩。

但这些问题在很多方面都太过简单了。数据大都取自权威来源，省去了大量准备工作。但在实际中，这样的数据集很少（可以自己指定的情况除外）。特别是现实中自带可用类标签的数据集少之又少。如果数据集中的标签不足，就无法构造出能精准预测验证集或测试集标签的分类器。此时应该怎么做呢？

通常会想到手动给数据贴标签，但这不仅耗时，还会产生一些特定的人工误差（这在高维数据集中很常见，因为人工观测无法像模型计算那样精准地识别类边界）。

一个比较新且有趣的备选方法是**半监督学习**。可以用该方法捕获潜在的数据分布状况来给无标签数据添加标签。半监督学习在过去十年间不断升温，因为用它能节省大量的标注时间，“标注”工作往往需要人类知识或专业设备。在自然语言解析和语言信号分析等领域，这项技术非常有价值，因为在这两大领域进行人工标注都非常复杂且耗时。

本章将介绍如何应用一些半监督学习技术，其中包括**对比悲观最大似然估计**（contrastive pessimistic likelihood estimation，CPLE）、自学习和 S3VM。可以在一系列复杂问题中运用这些技术给训练数据添加标签。借用一些近期在 scikit-learn 上名列前茅的 Python 库，将半监督技术应用于多个实例（包括音频信号数据），来展现半监督技术的功能和限制。

就此开始吧！

5.2 何为半监督学习

构造用于训练的标签数据的成本是机器学习中的一项固定成本。由于现实情况纷繁复杂，通常无法得到包含类标签的数据集，因此需要运用经过训练的分类技术来生成类标签，但在没有标

签训练数据和测试数据的情况下无法训练技术。如前所述，可以选择手动或通过测试给数据添加标签，但这种方法的时间成本、金钱成本（尤其对医疗测试而言）都过于高昂、难以组织，而且容易出错（如果数据集大而复杂的话）。半监督学习可以突破此困境。

半监督学习技术同时使用无标签数据和标签数据，这比仅使用无标签数据或标签数据的学习技术要好很多。除了监督学习（用标签数据）和无监督学习（用无标签数据），还有其他一些学习算法，包括半监督技术、转导推理技术和主动学习技术，等等。

半监督技术会预留一部分测试数据，不用于训练，而用于后期测试。转导推理技术能为无标签数据生成标签，该技术可能不包含测试过程，可能也没有可用的标签数据。

本章将重点介绍几种半监督技术，它们以常用格式提供了强大的数据标注功能。其中许多技术都可以用于包装常用的分类器，比如线性回归分类器或支持向量机。此外，许多技术都可以用 scikit-learn 的估计器运行。稍后先用线性回归分类器测试实例，然后应用半监督技术处理过的支持向量机。

5.3 半监督算法实战

前面简单介绍了半监督学习的概念、用途，以及一些半监督算法的应用现状，下面介绍如何高效应用半监督技术。

5.3.1 自训练

自训练是最简单的、最快的半监督学习方法，其应用广泛，包括自然语言处理和计算机视觉。实际上，自训练算法既能产生巨大价值，也会带来巨大风险。

自训练旨在将无标签实例的信息与标签实例的信息相结合，以迭代确定无标签实例的标签值。标签训练集会在每次迭代中增大，直到整个数据集都添加了标签。

自训练算法通常作为封装器应用于基础模型。本章将支持向量机作为自训练模型的基础模型。自训练算法非常简单，步骤较少，如下所示。

(1) 用一系列标签数据预测无标签数据（可以是所有无标签数据或部分无标签数据）的标签值。
(2) 计算所有新增标签实例的置信度。
(3) 从新增标签实例中挑选几个实例，用于下次迭代。
(4) 用所有标签实例（包括上次迭代挑选出的实例）训练模型。
(5) 重复(1)~(4)步，直到模型成功收敛。

整个过程如图 5-1 所示。

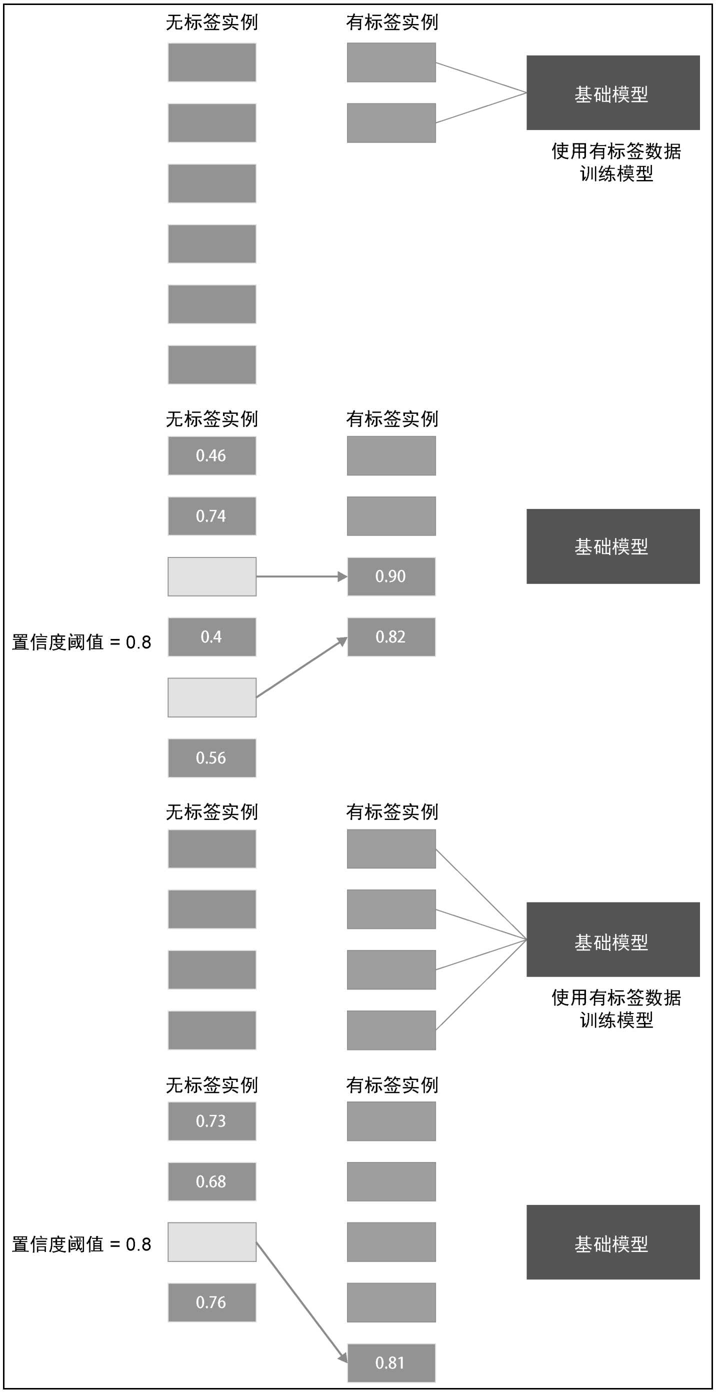

无标签实例
有标签实例
基础模型
使用有标签数据
训练模型
无标签实例
有标签实例
0.46
0.74
基础模型
0.90
置信度阈值 = 0.8
0.4
0.82
0.56
无标签实例
有标签实例
基础模型
使用有标签数据
训练模型
无标签实例
有标签实例
0.73
0.68
置信度阈值 = 0.8
基础模型
0.76
0.81

图　5-1

结束训练前，自训练模型应经过测试和验证。可以用交叉验证，或者使用其他特意保留的标签数据。

自训练非常强大，并且能够节约时间，但也存在风险。为了说明注意事项以及在分类算法中应用自训练的方法，下面详细介绍算法的运行过程。

本次实践将使用 semisup-learn GitHub 仓库的代码。在使用该代码前，需要先克隆相关 GitHub 仓库，相关指导见附录 A。

1. 实现自训练

自训练每次迭代的第一步是为无标签实例生成标签。首先构造一个 SelfLearningModel 类，并将基于监督算法的基本模型（basemodel）和迭代限制作为参数。稍后会讲到，迭代限制可以额外声明或指定为分类精度（即收敛程度）的函数。prob_threshold 参数提供接受标签的可信度的最低限制，低于该水平的预估标签都会被拒绝。除了硬编码阈值，后面的案例还涉及其他方法。

```
class SelfLearningModel(BaseEstimator):

def __init__(self, basemodel, max_iter = 200, prob_threshold = 0.8):
    self.model = basemodel
    self.max_iter = max_iter
    self.prob_threshold = prob_threshold
```

定义好 SelfLearningModel 类的外层后，接下来定义用于半监督模型拟合处理的函数。

```
def fit(self, X, y):
    unlabeledX = X[y==-1, :]
    labeledX = X[y!=-1, :]
    labeledy = y[y!=-1]

    self.model.fit(labeledX, labeledy)
    unlabeledy = self.predict(unlabeledX)
    unlabeledprob = self.predict_proba(unlabeledX)
    unlabeledy_old = []

    i = 0
```

参数 X 是输入数据矩阵，其形状等于[n_samples, n_features]，用于构建[n_samples, n_features]的矩阵。参数 y 是标签数组，无标签数据点在 y 中标记为-1。通过对 X 进行运算，即在 X 中选择 y 标签为-1 的元素，可以构建 unlabeledX 和 labeledX 参数。labeledy 参数在 y 上进行类似的选择。（通常我们对 y 中的无标签样本不感兴趣，但需要它们的标签来进行分类。）

标签预测的过程首先要用到 sklearn 封装的预测操作。unlabeledy 参数是用 sklearn 的 predict 方法生成的，predict_proba 方法则用于计算每个预测标签的概率。这些概率值存储在 unlabeledprob 中。

scikit-learn 的 predict 和 predict_proba 方法能够分别预测类标签和类标签的准确率。因为多个半监督算法会应用这两种方法，所以首先解释其原理。

predict 方法通过一系列二元分类器（即只区分两个类别的分类器）生成输入数据的类别预测。带有 n 个类别的完整模型包含多个二元分类器，具体如以下公式所示。

$$\frac{n*(n-1)}{2}$$

为了对给定实例进行预测，所有得分超过 0 的分类器都会给适合该实例的类标签投上一票，票数最多（而不是分类器总得分最高）的类别就是实例的类别。这是一种常用的方法，称作“一对一预测”。

同时，predict_proba 通过调用 Platt 校准（Platt calibration）运行。该技术可将分类模型的输出转换成类别的概率分布。首先训练基础模型，根据拟合的回归模型计算分类器的评分。

$$P(y\mid X)=\frac{1}{(1+\exp(A*f(X)+B))}$$

然后用最大似然方法优化该模型（通过调整参数 A 和 B）。在自训练模型的案例中，predict_proba 根据拟合的回归模型计算分类器的评分，从而得到每个类标签的概率。这非常有用！

接下来需要一个循环来进行迭代。以下代码使用了一个 while 循环，该循环执行直到 unlabeledy_old（unlabeledy 的副本）内无实例或达到最大循环次数。每次迭代时，尝试对每个没有标签、概率超过概率阈值（prob_threshold）的实例添加标签。

```
while (len(unlabeledy_old) == 0 or numpy.any(unlabeledy!=unlabeledy_old))
    and i < self.max_iter:
    unlabeledy_old = numpy.copy(unlabeledy)
    uidx = numpy.where((unlabeledprob[:, 0] > self.prob_threshold) |
                       (unlabeledprob[:, 1] > self.prob_threshold))[0]
```

然后，self.model.fit方法用模型拟合无标签数据，后者将以矩阵形式存储，矩阵大小为 [n_samples, n_features]（如前所述）。该矩阵会（通过vstack和hstack函数）不断补充无标签数据[①]。

```
self.model.fit(numpy.vstack((labeledX, unlabeledX[uidx, :])),
               numpy.hstack((labeledy, unlabeledy_old[uidx])))
```

最后预测标签，以及标签的概率。

① 即预测结果通过概率阈值的无标签数据。——译者注

```
unlabeledy = self.predict(unlabeledX)
unlabeledprob = self.predict_proba(unlabeledX)
i += 1
```

下次迭代时，模型将执行相同的处理，将概率预测超过阈值的新增标签数据作为 `model.fit` 步骤中的数据集的一部分。

如果模型还未包含能产生标签预测的分类方法（比如 sklearn 的支持向量机实现中的 `predict_proba` 方法），可以引用一个。以下代码查找 `predict_proba` 方法，并在找不到该方法时引用基于 `Platt scaling` 的标签生成函数。

```
if not getattr(self.model, "predict_proba", None):
    self.plattlr = LR()
    preds = self.model.predict(labeledX)
    self.plattlr.fit( preds.reshape( -1, 1 ), labeledy )

return self



def predict_proba(self, X):
    if getattr(self.model, "predict_proba", None):
    return self.model.predict_proba(X)
    else:
        preds = self.model.predict(X)
        return self.plattlr.predict_proba(preds.reshape( -1, 1 ))
```

准备就绪后，就可以开始应用自己的自训练架构了。先找个数据集试试吧！

以一个简单的线性回归分类器为基础模型（`basemodel`），其带有的**随机梯度下降**（stochastic gradient descent，SGD）为学习算法。输入数据集为 statlog 的 `heart` 数据集，取自 www.mldata.org，可从本章附带的 GitHub 仓库下载。

`heart` 数据集是一个二分类数据集，根据有无心脏病进行分类。数据集包含 270 个实例，每个实例有 13 个特征且没有缺失值。该数据没有标签，其中许多特征变量的标记成本很高而且需要经过很多复杂测试。具体变量如下所示。

- ❑ `age`
- ❑ `sex`
- ❑ `chest pain type (4 values)`
- ❑ `resting blood pressure`
- ❑ `serum cholestoral in mg/dl`
- ❑ `fasting blood sugar > 120 mg/dl`
- ❑ `resting electrocardiographic results (values 0,1,2)`
- ❑ `maximum heart rate achieved`
- ❑ `exercise induced angina`

- 10. oldpeak = ST depression induced by exercise relative to rest
- the slope of the peak exercise ST segment
- number of major vessels (0-3) colored by flourosopy
- thal: 3 = normal; 6 = fixed defect; 7 = reversable defect

下面分析 Heart 数据集。首先加载数据，然后用模型进行拟合。

```
heart = fetch_mldata("heart")
X = heart.data
ytrue = np.copy(heart.target)
ytrue[ytrue==-1]=0

labeled_N = 2
ys = np.array([-1]*len(ytrue)) # -1 denotes unlabeled point
random_labeled_points = random.sample(
    np.where(ytrue == 0)[0], labeled_N/2)+\random.sample(np.where(ytrue == 1)[0],
    labeled_N/2)
ys[random_labeled_points] = ytrue[random_labeled_points]

basemodel = SGDClassifier(loss='log', penalty='l1')

basemodel.fit(X[random_labeled_points, :], ys[random_labeled_points])
print "supervised log.reg. score", basemodel.score(X, ytrue)

ssmodel = SelfLearningModel(basemodel)
ssmodel.fit(X, ys)
print "self-learning log.reg. score", ssmodel.score(X, ytrue)
```

结果中规中矩。

```
self-learning log.reg. score 0.470347
```

多次尝试的输出质量波动极大（见图 5-2）。

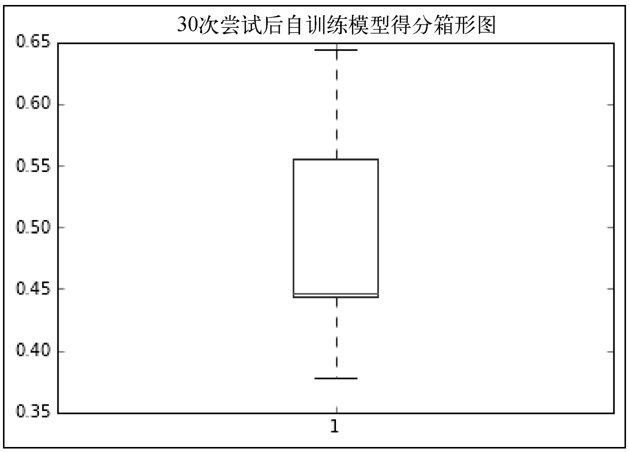


图　5-2

鉴于这是针对现实中的无标签数据进行分类精度评分，这个结果并不太糟，但也不够理想，因为仍有一多半的实例出现了分类错误。

该问题需要深入探究，因为现在仍不清楚哪里出了问题或者应该如何改进结果。下面回归到自训练的理论，了解如何诊断并改进实现。

2. 改善自训练实现

前面介绍了自训练算法，并尝试了其实现。然而第一次试验的成绩仍有待提高，精度和结果的波动性都存在问题，但足已表明自训练具有潜力。

自训练是个很脆弱的过程。如果算法的某个元素配置不当或输入数据含杂质，迭代过程就很可能一而再、再而三地出错，不断将标签错误的数据引入下一个添加标签的步骤中。因为自训练算法会迭代地自反馈，所以出现杂质较多的输入和输出是非常严重的问题。

一些常见风险值得一提。在一些情况下，标签数据可能并不会添加更多有用信息。这种情况在前几次迭代中很常见，也很好懂。通常最容易添加标签的无标签数据是与已有标签数据最为相似的那些数据。然而，因为给这些实例生成高概率标签很容易，所以这样做并不一定能降低接下来在迭代中添加标签的难度。

问题是，添加的实例有时对分类没有实际影响，而分类精度却整体下降了。更糟的是，添加与现有实例各方面相似的实例，以便其易于添加标签，这其实会误导分类器的决策边界，可能会使错误分类数增多。

有时很难判断自训练模型中的错误。如前所示，恰当地绘图有助于我们掌握情况。由于这种误差通常只在前几次迭代中出现，只要在标签预测循环中加入一个代表当前分类精度的变量，就能得知早期精度随迭代的变化了。

确定问题后，有几个可行的解决方案。如果有足够的标签数据，可以用一个更多样化的标签数据集来开始该过程。

自训练模型很容易出现过度拟合，为了应对这种风险，必须保留一些数据用于后续验证。另一种做法是用数据集的多个子集来训练多个自训练模型实例。这样经过多次尝试后，就能更加了解输入数据对自训练模型性能的影响。

第 8 章将介绍一些集成方法，以便用多个自训练模型共同生成预测。使用集成方法时，甚至可以考虑同时应用多种采样技术。

如果不想从数量上下手，可以考虑改进质量。方法之一是选择构建多样性适当的标签数据子集。开始自训练的标签实例没有最小数量等硬性限制。如果想从一个类中只有一个标签实例开始（像上个训练案例那样），很快就会发现，更多的标签数据有利于训练更多样化、重叠性更强的类。

自训练模型容易犯的另一类错误是偏差选择（biased selection）。本来设想每次迭代的数据选择最差只会出现一点偏差（即稍微偏向其中一类），但事实证明并非如此。许多因素都会影响偏差选择的可能性，罪魁祸首是对某一类的不合比例抽样。

如果整个数据集或使用的标签子集偏向某一类，自训练分类器过度拟合的风险就会增加。当实例传到下一次迭代，但其多样性不足以解决问题时，会使得问题更加复杂，自训练算法也会随之建立错误的决策边界，从而导致对数据子集过度拟合。各类的实例数量不均是问题关键，诊断与选择偏差相关的问题也是识别过度拟合的常用方法。

检验过度拟合的常用方法值得详述，因为这种技术（通常称作“验证技术”）颇具价值。验证技术的基本概念是使用两个数据集：一个用于构建模型，另一个用于测试模型。

最有效的验证技术是独立验证集检验，这是判断预测结果准确性的最简形式，但显然不是一直（或经常）能做到的。

因为可能没有单独的验证集，所以最好保留样本的一个子集。这种方法称作“样本分割”，是现代验证技术的基础。机器学习实现大都包括训练集、测试集和验证集，属于多层验证方法。

第三种也是最为关键的一种验证方法是重采样，即迭代地用数据子集来验证数据集。第 1 章就使用过 v 重交叉验证，而交叉验证可能是实践中最佳的重采样方法。

除了可用的技术，还要保证对数据进行有效建模所需的样本量。这没有普适原则，但可参考以下经验法则。

如果需要用 m 个点来确定单变量回归线，并达到一定精度，那么至少需要 mn 个观测值，甚至 $n!mn$ 个，才能恰当地描绘 n 个变量的回归函数的特性，并对其进行评估。

请注意，对该问题的推荐解决方案（重采样、样本分割以及交叉验证及其他验证技术）和前一个有些区别。也就是说，过度拟合要求限制标签训练数据子集的使用，更多的训练数据有助于避免模型初期便出错。对具体问题而言，所分析问题的复杂程度不同，需要达到的平衡也不同。通过观察问题出现时的迹象，以及时采取恰当的行动（即增加或减少在一次迭代中同时使用的标签数据量）。

自训练更严重的潜在风险是，无标签数据往往存在噪声。如果需要处理的数据集中的部分（甚至全部）无标签数据存在大量噪声，那么分类精度很可能会降低。

> 用数据复杂度和噪声测度来度量数据集中的噪声程度的做法由来已久，现有许多估计方法可用。
>
> 相对复杂度度量有两大类。其中之一度量的是不同类别的值的重叠区域，也称“可分性”，这种组内的度量能够描述每一类相对其他类的模糊程度。这种情况下，比较好的一种度量方法是最大**费舍尔判别比**（Fisher’s discriminant ratio），最大单个特征效率同样有效。
>
> 或者用线性分类器的误差函数来了解数据集内类别之间的分离程度（有时这样做更简单）。通过在数据集上训练一个简单的线性分类器，并观察训练误差，可以快速、全面地了解类别的线性分离程度。此外，与分类器相关的性能度量（比如处于类边界的样本占比，或类间/类内最邻近距离的平均比率）也很有帮助。
>
> 其他的数据复杂度测度可以专门度量数据集的密度或其他性质，例如最大覆盖范围比。再次强调，可以用线性分类器或包含非线性成分的分类器实现有效的度量。

3. 改良选择过程

自训练算法正常运行的关键指标是能精确计算每个预测标签的置信度。置信度的计算是自训练成功的关键。

第一次解释自训练时，给几个参数赋的值太过简单了，其中包括与置信度计算紧密相关的参数。选择标签实例时使用了一个固定的置信水平，用于比较预测概率，其实可以改用以下任意一种方法。

- 将所有预测标签加入标签数据集。
- 根据置信度阈值选出数据集中置信度最高的几个标签。
- 将所有预测标签加入标签数据集，并将置信度作为每个标签的权重。

总体而言，自训练的实现有很多问题，容易出现一系列训练失误，也容易过度拟合。更糟糕的是，随着无标签数据的增加，自训练分类器的精度也会越来越成问题。

下面考虑一个截然不同的自训练实现。虽然这种技术在概念上与前面介绍的算法非常相似，但它能在不同假设下得出不同结果。

5.3.2 对比悲观似然估计

前面对自训练技术的探索和应用表明，自训练既强大，又伴随着巨大的风险。因此需要使用诊断工具，对数据集也有严格要求。虽然可以通过采样、识别最佳标签数据，以及追踪一些数据集的表现来处理这些问题，但是对于一些标记成本高昂（比如医药或科学等需要专业知识和设备的领域）的数据来说，虽然自训练很适用，但前面讲的那些解决方法依旧很难实现。

有时最终得到的自训练分类器比监督分类器效果更好，这种情况其实很糟糕。更糟糕的是，基于标签数据训练的监督分类器会随着数据的增加而提升精度，有时半监督分类器却相反。因此，需要一种不那么“自以为是”的半监督学习方法。我们希望这种方法既具备半监督学习的优点，又能匹敌监督分类器的性能。

CPLE 是一种自监督学习方法，提供了一种更泛化的半监督参数估计方式。CPLE 有一个显著的优势：实践证明其标签预测精度超过其他半监督分类器，甚至包括基于标签数据构建的监督分类器。换句话说，在进行线性判别时，无须使用监督分析，可以用以 CPLE 为基础的半监督分析，后者的效果至少不会比前者差。

以上结论比较笼统，需要进一步证实。首先介绍一下 CPLE 的原理，然后在实际问题中验证其优越性。

CPLE 用最大对数似然函数来优化参数。下面即将构造的模型旨在通过优化模型参数来实现对数似然估计的最大化，可以视其为本模型的核心条件。CPLE 的一些保证和假设使得该技术高度可用。

为了构建更好的半监督学习器（基于监督学习器进行改良），CPLE 明确考虑了监督估计，并将半监督模型和监督模型间的损失作为训练性能测度（见图 5-3）。

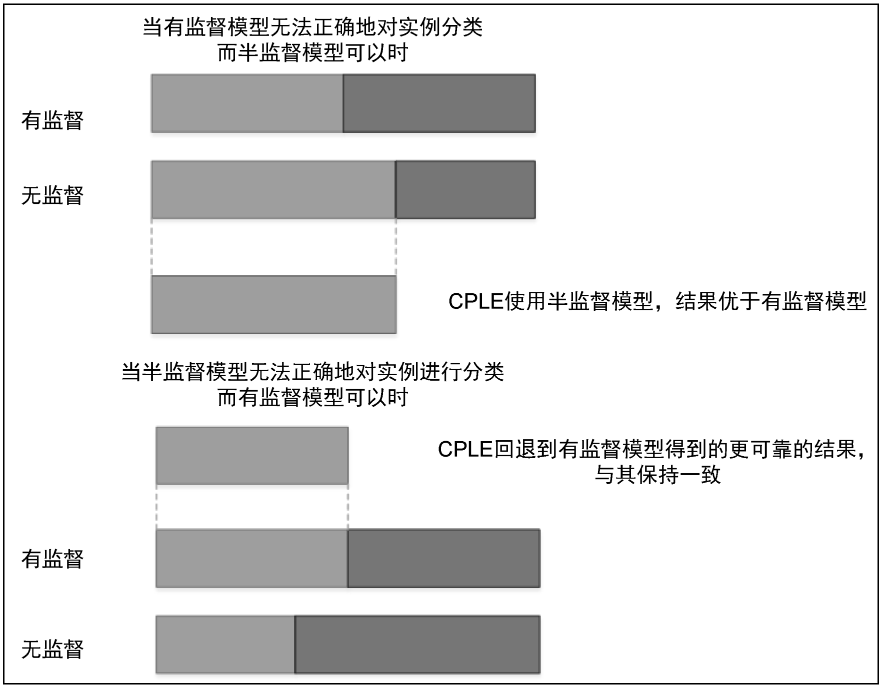


图　5-3

CPLE 计算所有半监督估计较监督解决方案的相对提升。如果监督解决方案更优，损失函数会如实显示，并通过训练调整半监督模型来降低损失。否则模型就会调整参数来学习半监督模型。

这听起来不错，但理论中存在一个不可忽视的瑕疵。半监督解决方案中不存在数据标签，这意味着无法取得后验分布（CPLE 用于计算损失）。因此 CPLE 采取了"悲观的"方法。CPLE 算法取所有标签/预测组合的**笛卡儿积**，然后选择能够使似然估计增益最小的后验分布。

在现实的机器学习环境中，该方法表现平稳。它能通过保守的假设提升半监督算法性能，从而确保监督方法的分类精度。在实际应用中，这些保守的假设能使得模型在测试时表现优良。不仅如此，CPLE 还可以在一些最具挑战性的无监督学习案例中进一步提升性能，这些问题中的标签数据往往无法很好地代表无标签数据（因为在一个或多个类别中的糟糕采样，或者单纯因为无标签实例不足）。

为了说明 CPLE 较半监督方法或监督方法的性能优势，下面用该技术解决一个实际问题。这次仍使用 semisup-learn 库，这个专业的 Python 库主要用于半监督学习，它拓展了 scikit-learn，以便为 scikit-learn 中的多个分类器提供 CPLE。首先构建一个 CPLE 类。

```
class CPLELearningModel(BaseEstimator):

    def __init__(self, basemodel, pessimistic=True, predict_from_probabilities =
        False, use_sample_weighting = True, max_iter=3000, verbose = 1):
        self.model = basemodel
        self.pessimistic = pessimistic
        self.predict_from_probabilities = predict_from_probabilities
        self.use_sample_weighting = use_sample_weighting
        self.max_iter = max_iter
        self.verbose = verbose
```

之前讲过`basemodel`的概念了。前面部署了S3VM和半监督的LDE，这里再次使用LDE[①]。首次尝试的目标是超过前面半监督LDE，结果也确实如此。

但在开始前，先查看一下其他参数选择。可以通过 `pessimistic` 参数应用非悲观（乐观）模型。`pessimistic` 方法会保守考虑无标签数据和标签数据的似然估计的差值，即把其最小化，而乐观模型会将其最大化，这样做结果更好（主要指训练集上的表现），但风险也更大。这里选择悲观模型。

`predict_from_probabilities` 参数通过考虑多个数据的预测概率来完成优化。设置该参数为 `true` 时，如果用于预测的概率超过均值，CPLE 会将该预测赋值为 `1`，反之为 `0`。还可以使用基础模型的预测概率作为预测结果，该指标性能更好，因此通常更受青睐。当然，对于大量实例，往往调用 `predict` 来实现批量预测。

① 此处的 S3VM 是指前文模型讲解阶段提及的以支持向量机为基准模型的半监督算法，而半监督 LDE 是前文模型应用阶段提及的以线性模型为基准模型的半监督算法。——译者注

还可以选用 use_sample_weighting，也称**软标签**（或“后验概率”）。通常会使用该参数，因为软标签比硬标签灵活，多作为首选（除非模型只支持硬标签）。

前几个参数用于判断 CPLE 训练何时停止，有的条件是达到最大迭代数，有的则是对数似然函数不再提升（通常是收敛导致的）。bestdl 记录最佳的似然估计差异值和对应的软标签，这些值会随每次训练迭代更新。

```
        self.it = 0
        self.noimprovementsince = 0
        self.maxnoimprovementsince = 3

        self.buffersize = 200
        self.lastdls = [0]*self.buffersize

        self.bestdl = numpy.infty
        self.bestlbls = []


        self.id = str(unichr(numpy.random.randint(26)+97))+str(unichr(numpy.random.
randint(26)+97))
```

discriminative_likelihood 函数计算给定输入下的似然估计（对于判别模型来说，即在给定输入 X 的情况下，寻找使 y=1 的概率最大的模型）。

> 请注意本例中的生成模型和判别模型的区别。虽然这不是一个基本概念，却是理解许多分类器目标的基础。
>
> 分类模型用输入数据对实例进行分类，并给每个实例分配标签。这样的方法有很多。
>
> 方法之一是在实例间绘制决策边界。随后获得新实例时，就能辨别它将落在哪一侧。这是一种**判别学习方法**。
>
> 另一种方法是单独模拟每一类的分布。模型生成后，算法就可以根据输入数据用贝叶斯法则计算标签的后验分布。这便是**生成方法**，它非常强大，但缺陷也显而易见（很大程度上与建模的性能相关）。生成方法包括高斯判别模型和许多贝叶斯模型。更多信息（包括一些优质的推荐文章）参见 5.4 节。

在本例中，该函数将在每次迭代时计算预测标签的概率。

```
    def discriminative_likelihood(self, model, labeledData, labeledy =
None, unlabeledData = None, unlabeledWeights = None, unlabeledlambda =
1, gradient=[], alpha = 0.01):
        unlabeledy = (unlabeledWeights[:, 0]<0.5)*1
        uweights = numpy.copy(unlabeledWeights[:, 0])

        uweights[unlabeledy==1] = 1-uweights[unlabeledy==1]

        weights = numpy.hstack((numpy.ones(len(labeledy)), uweights))
        labels = numpy.hstack((labeledy, unlabeledy))
```

定义好 CPLE 的这些部分后，还需要定义监督模型的拟合过程，这要用到 model.fit 和 model.predict_proba 组件来预测概率。

```
if self.use_sample_weighting:
    model.fit(numpy.vstack((labeledData, unlabeledData)),
              labels, sample_weight=weights)
else:
    model.fit(numpy.vstack((labeledData, unlabeledData)), labels)


P = model.predict_proba(labeledData)
```

为了执行悲观 CPLE，需要先根据监督数据和无监督数据分别计算判别对数似然值。下面依次在标签数据和无标签数据上执行 predict_proba。

```
try:

    labeledDL = -sklearn.metrics.log_loss(labeledy, P)
except Exception, e:
    print e
    P = model.predict_proba(labeledData)


unlabeledP = model.predict_proba(unlabeledData)



try:
    eps = 1e-15
    unlabeledP = numpy.clip(unlabeledP, eps, 1 - eps)
    unlabeledDL = numpy.average((
        unlabeledWeights*numpy.vstack((1-unlabeledy, unlabeledy)).T*numpy.log
        (unlabeledP)).sum(axis=1))
except Exception, e:
    print e
    unlabeledP = model.predict_proba(unlabeledData)
```

能够计算标签数据和无标签数据的判别对数似然值时，就可以通过 discriminative_likelihood_objective 函数设置目标了。这里的目标是用悲观方法（或乐观方法）计算每次迭代的 dl，直到模型收敛或达到最大迭代次数。

每次迭代时，用 t 检验判断概率是否发生变化。收敛前，概率应随每次迭代发生变化。如前所示，连续 3 次未变化的 t 检验意味着迭代应终止（可以通过 maxnoimprovementsince 参数配置次数）。

```
    if self.pessimistic:
        dl = unlabeledlambda * unlabeledDL - labeledDL
    else:
        dl = - unlabeledlambda * unlabeledDL - labeledDL
```

```
    return dl


def discriminative_likelihood_objective(self, model, labeledData,
    labeledy = None, unlabeledData = None, unlabeledWeights = None,
    unlabeledlambda = 1, gradient=[], alpha = 0.01):
    if self.it == 0:
        self.lastdls = [0]*self.buffersize

    dl = self.discriminative_likelihood(
        model, labeledData, labeledy, unlabeledData, unlabeledWeights,
        unlabeledlambda, gradient, alpha)

    self.it += 1
    self.lastdls[numpy.mod(self.it, len(self.lastdls))] = dl

    if numpy.mod(self.it, self.buffersize) == 0: # or True:
        improvement = numpy.mean((self.lastdls[(len(self.lastdls)/2):])) -
                                 numpy.mean((self.lastdls[:(len(self.
                                 lastdls)/2)]))

        _, prob = scipy.stats.ttest_ind(self.lastdls[(len(self.lastdls)/2):],
                                        self.lastdls[:(len(self.lastdls)/2)])

        noimprovement = prob > 0.1 and numpy.mean(
            self.lastdls[(len(self.lastdls)/2):]) < numpy.mean(self.lastdls
            [:(len(self.lastdls)/2)])
        if noimprovement:
            self.noimprovementsince += 1
            if self.noimprovementsince >= self.
            maxnoimprovementsince:

                self.noimprovementsince = 0
                raise Exception(" converged.")
        else:
            self.noimprovementsince = 0
```

每次迭代时，算法会存储最佳的判别似然值和权重集合，用于下次迭代。

```
if dl < self.bestdl:
    self.bestdl = dl
    self.bestlbls = numpy.copy(unlabeledWeights[:, 0])

return dl
```

如何构造软标签也值得探讨。前面探讨过这一话题，代码如下所示。

```
f = lambda softlabels, grad=[]: self.discriminative_likelihood_objective(
    self.model, labeledX, labeledy=labeledy,
    unlabeledData=unlabeledX,
    unlabeledWeights=numpy.vstack((softlabels, 1-softlabels)).T, gradient=grad)

lblinit = numpy.random.random(len(unlabeledy))
```

简而言之，softlabels 是判别似然值计算的概率版本，即它们是概率值，而非二元硬类标签。软标签用 optimize 方法计算。

```
try:
    self.it = 0
    opt = nlopt.opt(nlopt.GN_DIRECT_L_RAND, M)
    opt.set_lower_bounds(numpy.zeros(M))
    opt.set_upper_bounds(numpy.ones(M))
    opt.set_min_objective(f)
    opt.set_maxeval(self.max_iter)
    self.bestsoftlbl = opt.optimize(lblinit)
    print " max_iter exceeded."
except Exception, e:
    print e
    self.bestsoftlbl = self.bestlbls

if numpy.any(self.bestsoftlbl != self.bestlbls):
    self.bestsoftlbl = self.bestlbls
ll = f(self.bestsoftlbl)

unlabeledy = (self.bestsoftlbl<0.5)*1
uweights = numpy.copy(self.bestsoftlbl)

uweights[unlabeledy==1] = 1-uweights[unlabeledy==1]

weights = numpy.hstack((numpy.ones(len(labeledy)), uweights))
labels = numpy.hstack((labeledy, unlabeledy))
```


> 如有兴趣，可以了解一下优化过程用**牛顿共轭梯度**（Newton conjugate gradient）方法计算梯度下降，以寻找最佳权重值。5.4 节提供了牛顿共轭梯度的参考书目。

讲解完原理后，下面计算比较最佳监督标签和软标签，并将 bestsoftlabel 参数设为最佳标签集，然后用最佳标签集计算判别似然值，并计算 fit 函数。

```
if self.use_sample_weighting:
    self.model.fit(numpy.vstack((labeledX, unlabeledX)),
                   labels, sample_weight=weights)
else:
    self.model.fit(numpy.vstack((labeledX, unlabeledX)), labels)
if self.verbose > 1:
    print "number of non-one soft labels: ", numpy.sum(
        self.bestsoftlbl != 1), ", balance:", numpy.sum(self.bestsoftlbl<0.5),
        " / ", len(self.bestsoftlbl)
    print "current likelihood: ", ll
```

介绍过 CPLE 的实现后，下面用数据集试试吧。这次使用哥伦比亚大学的百万歌曲数据集。

该算法的主要特征是对 100 万首歌曲进行特征分析和数据诠释。这些数据经过了预处理，包含很多原始特征和衍生特征。可用特征包括歌手姓名和 ID、歌曲时长、响度、节拍和每首歌的

节奏，以及其他测度（包括大众对其作为舞曲的适合度评分以及音频相关标签）。

该数据集基本上自带标签，但这里的目标是基于数据生成不同歌曲的类型标签。因为该数据集非常庞大（300GB），所以只处理 1%，10 000 首歌曲（约 1.8GB）。此外，目前不需要数据的所有信息，其格式信息无用，许多变量也用不到。

`10000_songs` 数据集位于第 6 章的文件夹中，它是包含多个类型的音乐数据的子集，已经过清洗和预处理（但仍然很大）。本次分析将尝试根据作为目标变量的类型标签预测歌曲类型，将一部分标签作为标签数据，用于学习，然后为无标签数据生成标签。

本次迭代将以如下方式实现提升。

- 使用更多标签数据。这次随机选择整个数据集 1%的数据（100 首歌曲）作为标签数据。
- 用带有线性核的支持向量机作为分类器，而不是前面朴素自训练用过的简单线性判别分析。

开始吧！

```
import sklearn.svm
import numpy as np
import random

from frameworks.CPLELearning import CPLELearningModel
from methods import scikitTSVM
from examples.plotutils import evaluate_and_plot

kernel = "linear"

songs = fetch_mldata("10000_songs")
X = songs.data
ytrue = np.copy(songs.target)
ytrue[ytrue==-1]=0

labeled_N = 20
ys = np.array([-1]*len(ytrue))
random_labeled_points = random.sample(np.where(ytrue == 0)[0],
                                      labeled_N/2)+\
                        random.sample(np.where(ytrue == 1)[0],
                                      labeled_N/2)
ys[random_labeled_points] = ytrue[random_labeled_points]
```

为了进行比较，将同时运行监督支持向量机和 CPLE，并实现前面提到的朴素自监督算法。

```
basemodel = SGDClassifier(loss='log', penalty='l1') # scikit logistic regression
basemodel.fit(X[random_labeled_points, :], ys[random_labeled_points])
print "supervised log.reg. score", basemodel.score(X, ytrue)


ssmodel = SelfLearningModel(basemodel)
```

```
ssmodel.fit(X, ys)
print "self-learning log.reg. score", ssmodel.score(X, ytrue)


ssmodel = CPLELearningModel(basemodel)
ssmodel.fit(X, ys)
print "CPLE semi-supervised log.reg. score", ssmodel.score(X, ytrue)
```

本次迭代的结果非常不错。

```
# 有监督的逻辑回归得分：0.698
# 自学习逻辑回归得分：0.825
# CPLE 半监督逻辑回归得分：0.833
```

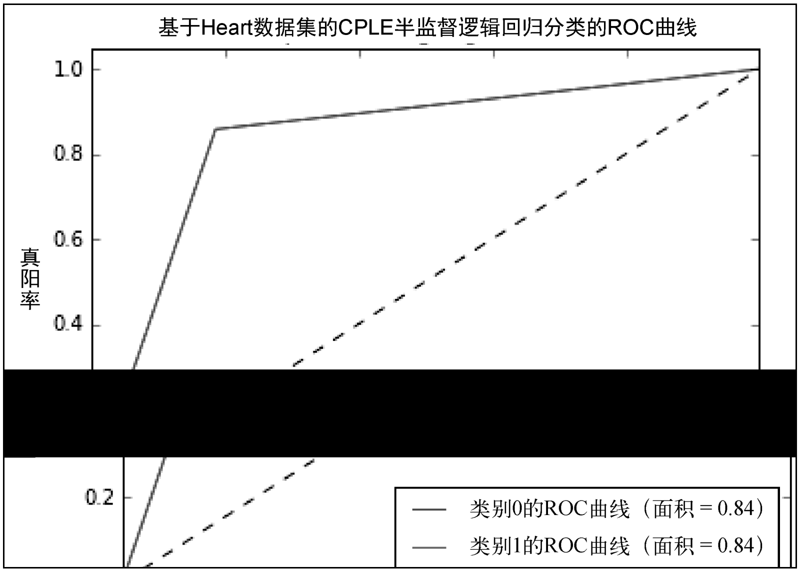


图 5-4

CPLE 半监督模型以 84%的精度成功完成了分类，该精度可与人类估计相媲美，且比朴素半监督实现高出约 10%。值得注意的是，该精度还超过了监督支持向量机。

5.4 扩展阅读

Xiaojin Zhu 撰写的“Semi-Supervised Learning Literature Survey”内容详尽，是理解半监督学习的良好起点。

该作者的另一篇文章也值得阅读：“Semi-Supervised Learning Tutorial”。

对于对比悲观最大似然估计，可参考 Marco Loog 在 2015 年发表的论文“Contrastive Pessimistic Likelihood Estimation for Semi-Supervised Classificatio”。

本章提到了生成模型和判别模型的区别，Andrew Ng（http://cs229.stanford.edu/notes/cs229-notes2.pdf）和 Michael Jordan（“Why the logistic function? A tutorial discussion on probabilities and neural networks”）对此做了相关明确说明。

如果对贝叶斯统计感兴趣，不妨阅读 Allen Downey 的 *Think Bayes*，这是一部入门佳作（我最喜欢的统计书之一）。

要想了解更多梯度下降知识，推荐阅读 Sebastian Ruder 的博客（http://sebastianruder.com/optimizing-gradient-descent/）。

如果想深入了解共轭下降，Jonathan Shewchuk 的文章“An Introduction to the Conjugate Gradient Method Without the Agonizing Pain”清楚而有趣地介绍了一系列关键概念。

5.5　小结

这一章介绍了机器学习中颇具实力但知名度不太高的领域——半监督学习。首先介绍了转导推理学习和自训练所涉及的概念，然后通过朴素自训练实现进一步展示了后一类技术。

自训练的缺陷很快就显露了，而 CPLE 可作为高效解决方案。CPLE 是一种优雅且高度可用的半监督学习框架，无须在用作基础模型的分类器上进行任何假设。CPLE 在大量朴素半监督和监督实现中得到了持续应用，且风险极小。本章详细介绍了机器学习中这项有用的新技术。

下一章将介绍数据处理技术，该技术可以有效提高模型的效率。

第 6 章

文本特征工程

6

6.1 介绍

前几章详细介绍了一些强大的技术，这些技术能够分析复杂、棘手的数据。然而，对于更复杂的问题，技术能做的始终有限。

一直以来，深度学习和监督学习寻找解决方案的过程往往需要团队投入大量成本。在旧模式下，通常需要进行特定的准备工作，这对专业技能和知识储备都有要求，还需要投入大量时间。此外，采用的技术往往仅针对特定领域或特定数据类型。这个生成特征的过程就称作**特征工程**。

前面介绍的大多数深度学习算法在尽量避免进行大量特征工程，但特征工程一直被视作顶尖机器学习从业者需要具备的一项重要技能。Kaggle 的顶尖参赛者在 Kaggle 的博客上发表了如下言论。

> “选用的特征对结果的影响远超其他元素。据我所知，没有任何单独的算法能够实现正确的特征工程所带来的信息增量。”
>
> ——Luca Massaron

> “毫无疑问，特征工程是 Kaggle 大赛最重要的方面之一，需要投入更多时间。数据中往往存在一些隐藏特征，这些特征能够大幅提升模型性能。要想名列前茅，就必须找到它们。如果在这方面表现不佳，基本上就没机会获胜了。除了找到所有隐藏特征，还有其他一些重要因素会影响排名，比如如何规划问题，是用回归模型还是分类模型，或者两者结合。这一点和特征工程都是在比赛中取得好成绩的关键。也有一些比赛不再要求（人工的）特征工程，比如图像处理比赛，因为目前先进的深度学习算法可以实现。”
>
> ——Josef Feigl

这里有几个关键点：特征工程非常强大，即使是极少量的特征工程，也能对分类器产生巨大影响，而且特征工程往往是取得最佳结果的关键因素。要想实现机器学习算法的性能最大化，需要具备一定的特定领域和特定数据类型知识。

还有一句话是：

“对于大多数 Kaggle 参赛者而言，特征工程是最重要的部分，而且易于学习。”

——Tim Salimans

Tim 所言不假，本章主要介绍直观且有效的数据处理技巧和数据转换方法，包括一些用于文本和时间序列数据的最有效且最常用的数据预处理方法。这些技术来自自然语言处理和金融时间序列相关应用。下面将介绍这些技术的原理、效果，以及如何诊断结果。

6.2　文本特征工程

前面介绍了一些从数据集中提取有用的特征子集的方法。这些通用方法在处理非数值型数据或非类型数据，以及无法简单转化成数值型数据或类型数据时，用处却不大。因此，处理文本数据时需要应用不同的技术。

本节将介绍的技术主要分为两类：数据清洗技术和特征准备技术。通常这两种技术按照顺序应用，下面依次介绍。

6.2.1　清洗文本数据

处理自然文本数据要用到一系列方法，因为现实中很少存在天然干净的文本数据。文本数据往往充满了拼写错误和表情符号等非字典结构，有时还有 HTML 标签，因此必须进行彻底清洗。

本节将使用许多高效的文本清洗技术来处理一个相当粗糙的实际数据集。该数据集是 2012 年 Kaggle 大赛的 Impermium 数据集，这次大赛的目标是构建能精确发现社交评论中含侮辱言论的模型。

没错，就是识别网络喷子。

下面开始吧！

1. 用 `BeautifulSoup` 进行文本清洗

首先人工检查输入数据。这一步非常关键，只有理解了文本数据中存在的问题，才能找到相应的清洗方式。

浏览这个充斥着网络仇恨言论的数据集真的很痛苦，因此这里只展示一例（见表 6-1）。

表　6-1

ID	日　期	评　论
132	20120531031917Z	"""\xa0@Flip\xa0how are you not ded"""

变量 ID 和日期似乎无须处理，但文本部分极具挑战性。本例中就出现了拼写错误和 HTML 元素。此外，通过添加空格或标点，数据集中的很多条目避开了脏话过滤。其他数据质量问题包括多重元音（为了扩充词语）、非 ASCII 码的字符、超链接……不胜枚举。

清洗该数据集的方法之一是在输入数据上执行正则表达式，处理数据质量问题。然而，庞大的数据量和多样化的问题形式使得使用正则表达式不太现实，至少一开始如此。可能因为错过了很多实例，或者误判了需要预处理的数据量，从而导致清洗过度或清洗不足。具体而言，可能会破坏真正的文本内容或遗留一些标签元素。解决方案是首先清除大部分常见的数据质量问题，以着重通过脚本处理剩余问题。

下面介绍 `BeautifulSoup`。这是一个非常强大的文本清洗库，还能移除 HTML 标记。下面用该库来处理钓鱼数据（见表 6-2）。

```
from bs4 import BeautifulSoup
import csv

trolls = []
with open('trolls.csv', 'rt') as f:
    reader = csv.DictReader(f)
    for line in reader:
        trolls.append(BeautifulSoup(str(line["Comment"]), "html.parser"))

print(trolls[0])

eg = BeautifulSoup(str(trolls), "html.parser")

print(eg.get_text())
```

表 6-2

ID	日　期	评　论
132	20120531031917Z	@Flip how are you not ded

如上所示，文本数据的质量大为改善，但这个例子也表明路还很长。如前所述，下面用正则表达式进行进一步清洗并标记数据。

2. 标点符号的处理和标记化

标记化（tokenisation）是将文本流拆分成记号序列的过程。记号可以是单词或字符集（比如表情符号或其他标点符号字符串，像???????? ）。

移除原始数据集中许多碍眼的 HTML 内容后，下面进一步清洗文本数据。为此，需要使用 `re` 模块，该模块利用正则表达式处理文本，比如字符串局部替换。下面在输入文本上进行一系列处理，主要包括用记号替换变量或有问题的文本元素，例如用记号`_EM` 替换所有 e-mail 地址。

```
text = re.sub(r'[\w\-][\w\-\.]+@[\w\-][\w\-\.]+[a-zA-Z]{1,4}', '_EM', text)
```

6

同样，用记号_U 替换 URL。

```
text = re.sub(r'\w+:\/\/\S+', r'_U', text)
```

还可以自动移除多余或有问题的空格、换行符、连字符和下划线。此外，首先处理多字符的问题，这种形式通常用于非正式对话中的强调。加长的一系列标点符号则编码为_BQ 和 BX。这些较长的标签能够与更直接的_Q 和_X 区别开来（二者分别指问号和感叹号）。

还可以用正则表达式处理多余的字母。将这样的字符串削减到至多 2 个字符，就能将组合数量减少到可控。削减后的组用_EL 标记。

```
# 将空格格式化
text = text.replace('"', ' ')
text = text.replace('\'', ' ')
text = text.replace('_', ' ')
text = text.replace('-', ' ')
text = text.replace('\n', ' ')
text = text.replace('\\n', ' ')
text = text.replace('\'', ' ')
text = re.sub(' +',' ', text)
text = text.replace('\'', ' ')


# 处理标点符号
text = re.sub(r'([^!\?])(\?{2,})(\Z|[^!\?])', r'\1 _BQ\n\3', text)
text = re.sub(r'([^\.])(\.{2,})', r'\1 _SS\n', text)
text = re.sub(r'([^!\?])(\?|!){2,}(\Z|[^!\?])', r'\1 _BX\n\3', text)
text = re.sub(r'([^!\?])\?(\Z|[^!\?])', r'\1 _Q\n\2', text)
text = re.sub(r'([^!\?])!(\Z|[^!\?])', r'\1 _X\n\2', text)
text = re.sub(r'([a-zA-Z])\1\1+(\w*)', r'\1\1\2 _EL', text)
text = re.sub(r'([a-zA-Z])\1\1+(\w*)', r'\1\1\2 _EL', text)
text = re.sub(r'(\w+)\.(\w+)', r'\1\2', text)
text = re.sub(r'[^a-zA-Z]','', text)
```

然后继续构建其他需要的记号，其中最有用的是指代辱骂的_SW 记号。下面用正则表达式来辅助识别微笑符号，并将其标记为以下四类之一：开心的大笑（_BS）、开心的微笑（_S）、十分难过（_BF）和有点难过（_F）。

```
text = re.sub(r'([#%&\*\$]{2,})(\w*)', r'\1\2 _SW', text)


text = re.sub(r' [8x;:=]-?(?:\)|\}|\]|>){2,}', r' _BS', text)
text = re.sub(r' (?:[;:=]-?[\)\}\]d>])|(?:<3)', r' _S', text)
text = re.sub(r' [x:=]-?(?:\(|\[|\||\\|/|\{|<){2,}', r' _BF', text)
text = re.sub(r' [x:=]-?[\(\[\|\\/\{<]', r' _F', text)
```

> 微笑符号非常复杂，因为它们的用法多变。虽然目前这一系列字符相当常用，但不够全面。比如可以从表情符号中看到一些非 ASCII 码的表示方式。出于一些原因，需要删除本例中的非 ASCII 码文本（另一种类似的方法是通过一个词典强制规范文本），这两种方法都存在明显缺陷，因为都删除了数据集中的实例，即二者都不完美。在一些情况下，这种方法可能会导致删除数据过多。总而言之，应意识到文本领域中存在文字组合而成的图像是常见问题。

然后将文本分割成短语。这一步应用了 `str.split`，该函数能将输入作为单词向量进行处理，而不是长字符串（`re`）。

```
phrases = re.split(r'[;:\.()\n]', text)
phrases = [re.findall(r'[\w%\*&#]+', ph) for ph in phrases]
phrases = [ph for ph in phrases if ph]

words = []

for ph in phrases:
    words.extend(ph)
```

结果如表 6-3 所示。

表 6-3

ID	日 期	评 论
132	20120531031917Z	[['Flip', 'how', 'are', 'you', 'not', 'ded']]

然后搜寻单个字母的序列。有时网络上的讨论会以空格隔开的单字母串来强调某些事情，因为这种做法能够避开脏话检测。


```
tmp = words
words = []
new_word = ''
for word in tmp:
    if len(word) == 1:
        new_word = new_word + word
    else:
        if new_word:
            words.append(new_word)
            new_word = ''
        words.append(word)
```

前面做了许多工作来清洗输入数据和提升其质量。但数据中仍有问题未解决。下面重新思考一下最初这个案例，该案例目前如表 6-4 所示。

表 6-4

ID	日 期	评 论
132	20120531031917Z	['_F', 'how', 'are', 'you', 'not', 'ded']

经过前面的清洗，可以看到 HTML 标签清洗和向量化处理对句子内容的影响，表情符号也很好地替换为了_F 记号。表 6-5 所示的稍复杂的案例展示了文本处理前后的显著改变。

表　6-5

原　文　本	清洗和拆分后
GALLUP DAILY\nMay 24-26, 2012 \ u2013 Updates daily at 1 p.m. ET; reflects one-day change\nNo updates Monday, May 28; next update will be Tuesday, May 29.\nObama Approval48%-\nObama Disapproval45%-1\nPRESIDENTIAL ELECTION\nObama47%-\nRomney45%-\n7-day rolling average\n\n It seems the bump Romney got is over and the president is on his game.	['GALLUP', 'DAILY', 'May', 'u', 'Updates', 'daily', 'pm', 'ET', 'reflects', 'one', 'day', 'change', 'No', 'updates', 'Monday', 'May', 'next', 'update', 'Tuesday', 'May', 'Obama', 'Approval', 'Obama', 'Disapproval', 'PRESIDENTIAL', 'ELECTION', 'Obama', 'Romney', 'day', 'rolling', 'average', 'It', 'seems', 'bump', 'Romney','got', 'president', 'game']

以上两个案例仍明显存在两个严重问题。首先，第一个案例中存在一个拼写错误的单词，需要设法删除它。其次，两个案例中的许多单词（比如 are 和 pm）本身意义不大。还有一个问题，尤其是对短文本样本来说，清洗过后的内容可能只包含一两个有意义的词语。如果这些词语在语料库中并不常见，那么很难训练分类器来识别这些词语的含义。

3. 单词的标记和分类

英语单词按词性（也称“词类”）可分为名词、动词、副词，等等。如果知道某个单词是形容词，而不是动词或停止词（比如 a、the、of），算法就能有针对性地进行处理。

如果能识别词性并将其编码为类型变量来对词语进行分类，就能只保留有价值的内容来提高数据质量。文本标记的方式和技巧多种多样，一节无法尽述，因此只介绍部分可应用的标记技术：*n*-gram 标记和回退标记，这对技术互补，有助于构建强大的递归标记算法。

可以使用 Python 的 NLTK（natural language toolkit，自然语言工具）库。NLTK 功能丰富，本章后面很多地方也会用到。下面先用 NLTK 标记和删除几类词，此处特指停止词。

那么问题来了——为什么要删除停止词？通常停止词对大部分文本分析影响极小或毫无影响，反而会带来一定程度的噪声和训练方差。删除停止词的方法相当简单。加载 NLTK，下载并载入词典，然后对之前的单词向量进行全局扫描，将找到的所有停止词删除即可。

```
import nltk
nltk.download()
from nltk.corpus import stopwords

words = [w for w in words if not w in stopwords.words("english")]
```

该步骤很简单。下面探索 NLTK 的更多功能，特别是标记（tag）。

- **用 NLTK 标记**

如前所述，标记是识别词性、给每个词添加标签的过程。

最简单的标记方法和在输入数据上应用词典一样简单，就像处理停止词那样。

```
tagged = ntlk.word_tokenize(words)
```

显然，实际使用的语言更为复杂。一个词（例如 ferry）可能拥有多种词性，无法轻易确定其在每个语境中的词性。大多数情况下，只有给出上下文和单词在短语中的位置，才能正确理解标签的含义。

有多种技术可解决语言问题。

4. 序列标记

序列标记算法从左到右逐词（因而称为“序列”）遍历输入数据集，给每个词添加标签。所添加的标签类型取决于该词、该词前面的词，以及前面词的预测标签。

下面使用 ***n*-gram 标记器**。这是一种序列标记，经过预训练后能够识别特定标签。*n*-gram 标记器能根据该单词和前面 $n-1$ 个词性标签（POS tag）生成标签。

> 请注意，*n*-gram 是指给定元素集中 *n* 个元素的连续序列。元素可以是字母、单词、数字码（比如状态变换）或其他元素。*n*-gram 广泛用于捕捉 *n* 个元素（短语或编码的状态变换）的联合含义。

n-gram 标记器的最简形式是取 $n=1$，即**一元标记**。一元标记器会统计每个记号的条件频率分布。这里的条件频率分布通过训练术语语料库构建，而这种训练可以用 NLTK 中 `NgramTagger` 类的训练方法实现。该标记器假设，给定序列中某个记号出现最频繁的标签很可能就是该记号的最恰当标签。如果 carp 一词在一段序列中 4 次为名词、2 次为动词，那么一元标记器就会给类别为 carp 的所有记号加上名词标签。

以上做法对初步标记而言足够了，但只给每个同音词添加一个标签显然远远无法满足要求。需要增大 *n*-gram 中的 *n*。比如当 $n=3$（即用**三元标记器**）时，标记就能更轻松地辨别“He tends to carp on a lot”（他特别吹毛求疵）和“He caught a magnificent carp”（他得到了一条漂亮的鲤鱼）。

标记准确度和标记能力之间仍需要权衡。当 *n* 增大时，*n*-gram 的长度也会增大，同时也会越来越罕见。很快数据中将不再出现 *n*-gram，标记器也就无法再为现有记号寻找恰当标签了。

显然，需要运用一系列标记器。先让最准确的标记器给数据集贴标签，如果失败了，再用更可靠，但准确度可能略低的标记器进行尝试。

现有的回退标记足以满足需求了，下面详述。

- **回退标记**

有时给定的标记器可能不够可靠，尤其当准确度要求较高，而训练数据有限时。此时通常要

构建一个集成结构，同时使用多个标记器。

这里需要区分两类标记器：**下标器**和**回退标记器**，前者类似于前面讲过的**序列标记器**和 **Brill 标记器**。标记结构可能会包含一种或多种标记器。

如果下标器无法确定某个记号的标签，可以转而使用回退标记器。回退标记器专门用于合并下标器组合（一或多个下标器）的结果，如图 6-1 所示。

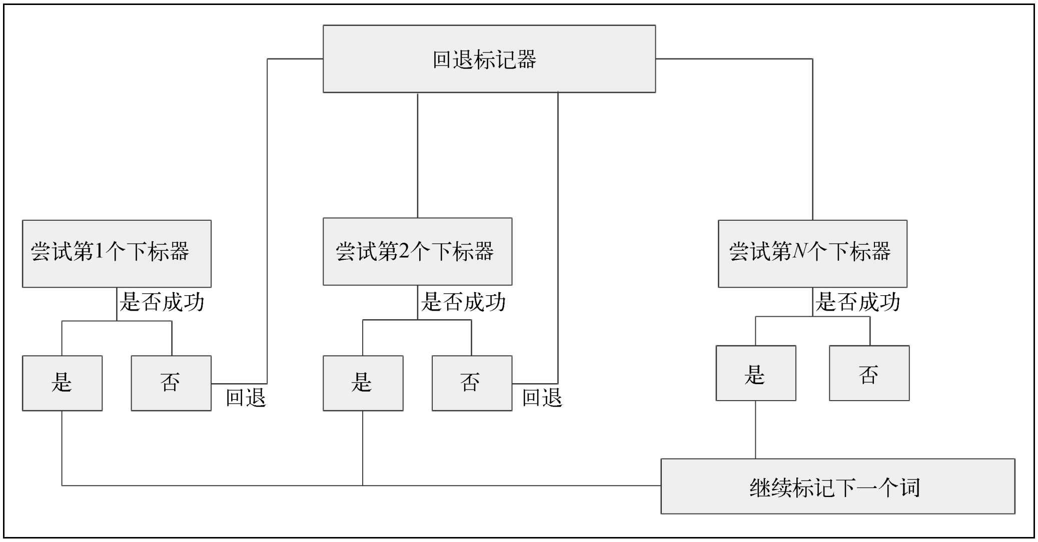


图 6-1

简单情况下，回退标记器会给下标器按序投票，并接受第一个非空值的标签。如果给定记号的所有下标器都为空值，那么回退标记器会给该记号添加“无”的标签。下标器的顺序可以自定义。

回退标记往往与其他下标器一同使用，以便数据科学家利用多种标记器的优势。根据具体需求，回退有多种方式，还可以创建高度冗余或极其复杂的标记结构（见图 6-2）。

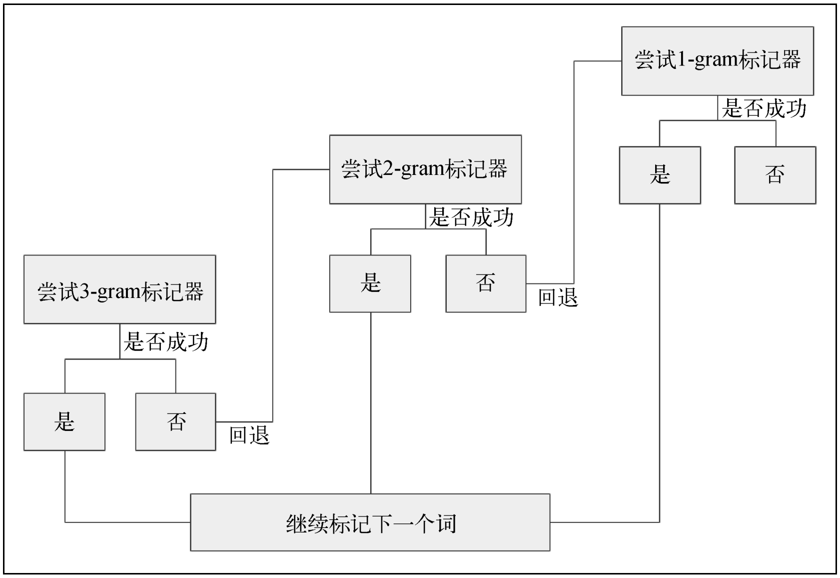


图 6-2

通常回退标记器会提供冗余，便于在组合解决方案中使用多个标记器。为解决眼下这个问题，需要实现一组嵌套式的 *n*-gram 标记器。首先将二元标记器作为回退标记器的三元标记器。如果这些标记器都没有解，再将一元标记器作为额外的回退标记。做法很简单，如下所示。

```
brown_a = nltk.corpus.brown.tagged_sents(categories= 'a')

tagger = None
for n in range(1,4):
    tagger = NgramTagger(n, brown_a, backoff = tagger)

words = tagger.tag(words)
```

6

6.2.2 根据文本数据构造特征

文本清洗过后，还要经过一些处理，将文本变为有用的特征。对此，要用到自然语言处理中的以下重要技术。

- 词干提取
- 词性还原
- 基于随机森林的 bagging 算法

1. 词干提取

处理语言数据集的另一个棘手问题是，词干存在多种形式。比如，词根 dance 就是许多单词

的词干：dancing、dancer、dances 等。如果能将这些形式简化为词干，就能提升 *n*-gram 的标记效果，也能应用一些新技术，比如词性还原。

能将单词简化成词根形式的技术叫作“词干提取器”。它可以将单词解析为辅音/元音字符串，并应用一系列规则。最常用的词干提取器是**波特词干器**（porter stemmer），其执行步骤如下所示。

(1) 简化后缀的范围，将其缩小到更小的集合（例如将 ies 变为 i）。

(2) 用多种方式删除后缀，每种方式都会删除一系列后缀（比如过去分词后缀或复数后缀，例如 ousness 或 alism）。

(3) 删除后缀后，给需要的单词末尾添加 e（比如 ceas 变成 cease），以整理词尾。

(4) 删除成对出现的 l。

波特词干器算法非常高效。下面通过实际应用来说明。

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

stemmer.stem(words)
```

前面的案例展示过，该 `stemmer` 的输出是单词的词根形式。这可能是个真实的单词，也可能不是，比如 dancing 变成了 danci。这样虽无伤大雅，但也不够理想，或许可以做得更好。

为了得到真实单词，需要运用稍微不同的技术——“词性还原”。词性还原判断单词词根的过程更复杂。与波特词干器不同，词性还原对不同的词性执行不同的标准化过程，而且会寻找单词的真正词根。词干可能不是真实单词，但词根一定是。词性还原还能将同义词简化成词根形式。比如词干提取可能会将 books 一词简化为 book，而无法处理 tome。而词性还原能够将二者都简化为 book。

每个输入记号的 POS 标记是一项必要前提，前面已经实现了 POS 标记器，可以直接处理该过程的输出结果了。

```
from nltk.stem import PorterStemmer, WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

words = lemmatizer.lemmatize(words, pos = 'pos')
```

这样的输出是理想的形式了（见表 6-5）。

表　6-5

源　文　本	词干提取处理后
The laughs you two heard were triggered by memories of his own high-flying exits off moving beasts	['The', 'laugh', 'two', 'hear', 'trigger', 'memory', 'high', 'fly', 'exit', 'move', 'beast']

至此，成功提取了输入文本数据的词干，大幅提升了查找算法（比如许多基于词典的方法）在处理该数据时的表现，也删除了停止词，用正则表达式方法给一些噪声元素做了记号，还删除了所有 HTML 标签。文本数据已经处理成合理可用的状态了，但还需要应用一项关键技术以从文本数据中生成特征。具体而言，可以用 bagging 量化词语的使用。

继续探索吧！

2. bagging 和随机森林

bagging 是子空间方法中的一种技术。该方法有多种形式，每种形式都有单独的名字。如果要从数据集中随机提取子集，可用 pasting；如果要从可放回的数据集中抽样，可用 bagging；如果不是从数据集中抽样，而是选取特征的子集，可以执行“特征 bagging”；如果既从数据集中抽样，又从特征中抽样，那么可用**随机补丁**技术。

基于特征的技术、特征 bagging 和随机补丁方法都能在特定场景中发挥巨大作用，尤其当数据维度极高时。医药和基因领域常常包含大量维度极高的数据，因此基于特征的方法适用于这些领域。

自然语言处理中，最常用的是 bagging。在语言数据领域，将需要处理的对象称作“词袋”（bag of words）。词袋法是一种文本数据准备方法，可以识别数据集中所有不同的词（或记号），并计算其在每个样本中出现的频次。下面以数据集中的案例来说明（见表 6-6）。

表 6-6

ID	日 期	评 论
132	20120531031917Z	['_F', 'how', 'are', 'you', 'not', 'ded']
69	20120531173030Z	['you', 'are', 'living', 'proof', 'that', 'bath', 'salts', 'effect', 'thinking']

得到了包含 13 个元素的词语列表，如下所示。

```
[
  "_F"
  "how"
  "are"
  "you"
  "not"
  "ded"
  "living"
  "proof"
  "that"
  "bath"
  "salts"
  "effect"
  "thinking"
]
```

使用该列表的下标，可以对前面每个句子构建出含 13 个元素的向量。向量值为遍历整个列表后每个词在数据集中对应句子中出现次数的计数。根据之前的示例句子以及根据它们构建的列表，最后构建出了表 6-7 所示的词袋。

表　6-7

ID	日　　期	评　　论	词　　袋
132	20120531031917Z	_F how are you not ded	[1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
69	20120531173030Z	you are living proof that bath salts effect thinking	[0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1]

这就是词袋法实现的核心。通常将文本中的语言内容转换成数值向量后，就能用一些技术灵活地对文本进行分类了。

可以给词语赋以权重。通过对词语加权调整向量值，可强调具有象征意义或有助于分类的词。加权可以是简单的掩码，比如指示是否存在的二元掩码。

二元掩码适用于某些词语的词频远高于平均水平的情况。此时，如果不使用二元掩码，就要进行特定的标准化（例如对数标准化）。但词频往往也带有信息（比如表示强调），因此很难轻易决定是否要用二元掩码。

另一种加权方法是词频—逆文档频率（term frequency-inverse document frequency，TF-IDF）。该方法会比较单词在特定句子和整个数据集中的词频，如果单词在给定样本中出现的频率高于在整个语料库中出现的频率，那么该词的权重值会增大。

TF-IDF 的变体常用于文本挖掘，例如搜索引擎。scikit-learn 提供了 TF-IDF 实现 `TfidfVectoriser`，稍后将用该技术部署 TF-IDF。

前面介绍了词袋法的理论知识，以及在构建好单词向量后可采用的一系列后续技巧，下面探讨如何实现词袋法。词袋法可用作模型的封装器。通常子空间法可以使用任何基础模型（多是支持向量机和线性回归模型），但词袋法实现常用到随机森林，并且往往将数据准备和学习过程封装到一个简洁的脚本中。下面先实现词袋法，稍后实现基于随机森林的分类算法。

> 第 8 章（该章将介绍可构建的多种集成方法）将详细介绍随机森林，但现在有必要说明，随机森林是一系列决策树。它们是强大的集成模型，可以并行（产生投票或其他结果）或相互提升（迭代地添加新树，对解决方案中现有树集不能很好建模的部分进行建模）。
>
> 由于随机森林强大且易用，所以常用作基准算法。

词袋法的实现过程也很简单。首先初始化 bagging 工具（即向量化器）。请注意，本例要限制特征向量的大小，以节省时间，因为每篇文本必须与特征列表中的每一项进行比较，所以在运行分类器时，该过程会比较耗时。

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(analyzer = "word",   \
                             tokenizer = None,    \
                             preprocessor = None, \
                             stop_words = None,   \
                             max_features = 5000)
```

然后通过 `fit_transform` 用向量化器拟合单词数据。将数据转化成特征向量是拟合过程的一部分。

```
train_data_features = vectorizer.fit_transform(words)

train_data_features = train_data_features.toarray()
```

至此就完成了文本数据的预处理。前面用一系列文本挖掘技术处理了该数据集，介绍了每种技术背后的理论，并用一些强大的 Python 脚本处理了测试数据集。下面探索 Kaggle 的侮辱检测问题。

6.2.3 测试准备好的数据

完成数据集的初步准备后，下面尝试处理实际问题，以检验自身水平。Impermium 的说明和数据描述如下。

> 此为单类分类问题，标签为 0 代表中立评论，为 1 代表侮辱性言论（不将“中立”视作侮辱）。预测值必须落于[0, 1]，其中 1 代表 100%确定该评论是侮辱性言论。
>
> - 寻找大型博客或论坛中对用户有侮辱倾向的言论。
> - 而不是针对非用户（比如明星、公众人物等）的侮辱言论。
> - 侮辱评论包括亵渎、种族歧视或其他攻击性语言，但这种情况很少。
> - 涉及亵渎或种族歧视，但没有确定的侮辱对象的言论，不算侮辱言论。
> - 言论的侮辱性必须很明显，而非不易察觉。
> - 标签中存在少量未清洗掉的噪声。但参赛者可以放心，训练数据和测试数据中的误差不超过 1%。
>
> 参赛者还应知悉，本问题可能会出现严重的过度拟合。所提供的数据一般能代表整个测试数据集，但并不详尽。Impermium 将从海量样本中提取部分未公开数据，并基于此进行最终评估。

这段说明很不错，它强调的两点值得注意。需要的评测指标是 AUC（area under the curve），该指标对“假阳性”和“假阴性”结果（即“特异性”和“灵敏性”）都很敏感。

这段说明明确表示，连续型预测值比二元（0/1）输出更符合要求。这一点在使用 AUC 时尤为重要。如果使用的是类型变量值，即使错误预测极小，也会导致分数骤降。因此，不应使用

`RandomForestClassifier` 算法，而应使用基于回归的 `RandomForestRegressor` 算法，然后重新将结果标准化到 0~1。

实际的 Kaggle 比赛更具挑战性，也更真实，那里没有现成的正确解决方案。第 8 章将介绍顶级数据科学家如何应对此类场景并取得成功。这里将利用测试集评测模型的表现。注意，这种优势也伴随着风险，如果出现过度拟合的问题，则必须遵守规则，确保并没有过度拟合测试集。

第 8 章将讨论实际参赛者的成果。但可以想见，名列前茅的参赛者会有很多失败的尝试，因此应设定一个基准，以帮助确定模型是否在朝正确的方向发展。

具体而言，个人（测试）排行榜上排名前 14 的参赛者的 AUC 都超过了 0.8，第一名更是达到了惊人的 0.84，而 50 支团队多半分数超过了 0.77。

如前所述，先构建随机森林回归模型。

> 随机森林是决策树的集成。
>
> 单独一棵决策树可能会出现方差或偏差问题，而随机森林会对多个并行试验进行加权平均，以均衡各种建模的结果。
>
> 随机森林易于使用。对于新的数据问题，可考虑先应用该方法。在前期对数据应用随机森林分类器，有助于大致了解最初的分类精度的基准水平，也有助于掌握分类边界的形成过程。在处理数据集的初级阶段，这样的洞悉非常重要。
>
> 借助 scikit-learn 的 `RandomForestClassifier`，可以轻松应用随机森林算法。

第一次将用 100 棵树，虽然增加树的数量会提升分类精度，但耗时也更久。通常构建模型的初期最好进行快速迭代。重复运行模型的速度提高，有助于快速了解并改进模型结果。

首先，初始化模型并进行训练。

```
trollspotter = RandomForestRegressor(
    n_estimators = 100, max_depth = 10, max_features = 1000)

y = trolls["y"]

trollspotted = trollspotter.fit(train_data_features, y)
```

然后将模型应用于测试数据，以预测每个测试实例的分数。再用简单的拉伸技术对这些分数进行标准化。

```
moretrolls = pd.read_csv('moretrolls.csv', header=True,
                         names=['y', 'date', 'Comment', 'Usage'])
moretrolls["Words"] = moretrolls["Comment"].apply(cleaner)

y = moretrolls["y"]
```

```
test_data_features = vectorizer.fit_transform(moretrolls["Words"])
test_data_features = test_data_features.toarray()

pred = pred.predict(test_data_features)
pred = (pred - pred.min())/(pred.max() - pred.min())
```

最后，应用 roc_auc 函数来计算模型的 AUC。

```
fpr, tpr, _ = roc_curve(y, pred)
roc_auc = auc(fpr, tpr)
print("Random Forest benchmark AUC score, 100 estimators")
print(roc_auc)
```

可以看到，最后的结果远未达到要求。

```
Random Forest benchmark AUC score, 100 estimators
0.537894912105
```

合理配置以下选项有助于提高模型质量。

- 处理输入数据的方法（包括数据预处理阶段和数据标准化阶段）。
- 随机森林中的评估器数目。
- 所部署的分类器类型。
- 实现的词袋法的属性（尤其是词语的最大数量）。
- *n*-gram 标记器的结构。

再次尝试时，调整词袋法实现的词语规模，将词语规模从随机设置的 5000 提升到 8000 左右，并且不只选择一个值，而是在一个范围内运行，看看能学到什么。树的数量也将提升至更合理的数值（该数值在本例中为 1000）。

```
Random Forest benchmark AUC score, 1000 estimators
0.546439310772
```


结果比之前稍好，但提升幅度并不明显，离目标依旧甚远。下面更进一步，建立另一个分类器——支持向量机。首先建立自己的支持向量机对象。

```
class SVM(object):

    def __init__(self, texts, classes, nlpdict=None):

        self.svm = svm.LinearSVC(C=1000, class_weight='auto')
        if nlpdict:
            self.dictionary = nlpdict
        else:
            self.dictionary = NLPDict(texts=texts)
        self._train(texts, classes)

    def _train(self, texts, classes):
        vectors = self.dictionary.feature_vectors(texts)
        self.svm.fit(vectors, classes)
```

```
    def classify(self, texts):
        vectors = self.dictionary.feature_vectors(texts)
        predictions = self.svm.decision_function(vectors)
        predictions = p.transpose(predictions)[0:len(predictions)]
        predictions = predictions / 2 + 0.5
        predictions[predictions > 1] = 1
        predictions[predictions < 0] = 0
        return predictions
```

虽然 SVM 还达不到人类评估水准，但作为一种算法，它运行高效，能迭代地将数据集转化成多个额外维度，从而在最优类边界上构造复杂的超平面。因此，分类质量的提升也在意料之中。

```
SVM AUC score
0.625245653817
```

要想了解结果如何，可以尝试用不同的方法对性能进行测度。这里要特别关注模型的标签预测和实际目标的差异，以观察模型是否在特定种类的输入上更容易失败。

前面做了不少预测。虽然还有不少技术可用，但不妨考虑将更复杂的集成模型作为一种可靠选择。本例利用多个模型（而非单一模型）能取得每个模型的相对优势。若想试试集成模型在本例上的效果，可以运行 score_trolls_blendedensemble.py 脚本。

> 该集成模型是混合的（也称“堆叠的”），第 8 章将深入介绍该集成模型的原理。

将结果绘制成图像，可以看到性能有所提升，但未达到期望（见图 6-3）。

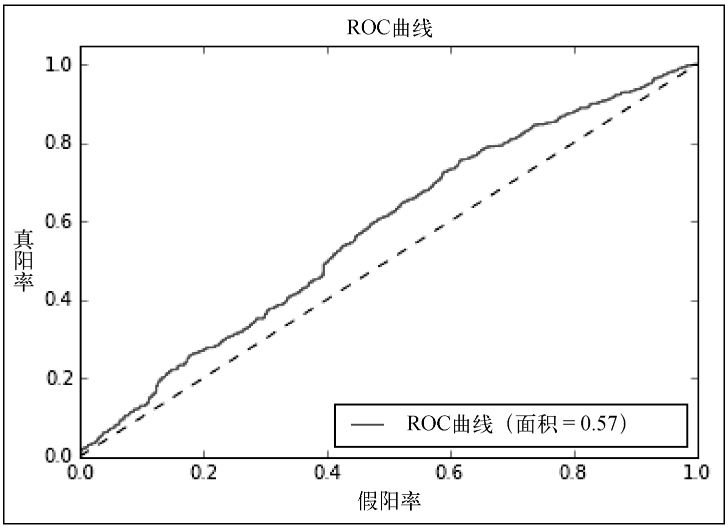


图　6-3

显然，构建模型时出了一些问题，但现在无须再构建一个更成熟的模型，而应该关注特征，并扩展特征集。

Kaggle 大赛的参赛者身上有一些东西值得学习。总体而言，排名靠前的模型往往发掘了输入数据的所有特征。Kaggle 大赛中对该数据集进行分类的第二名是 tuzzeg，他在 https://github.com/tuzzeg/detect_insults 提供了可用的代码库。

相比于我们的模型，tuzzeg 的模型更加细致全面。除了用 POS 标记构造的基础特征外，tuzzeg 还应用了基于 POS 的二元标记、三元标记，等等（通过在 *n* 个单词上滑动窗口构建）。tuzzeg 总共用到了七元标记，构造了长度为 2、3、4 的 *n*-gram。

此外，tuzzeg 还构造了两种复合模型：句子层次模型和排序模型，每种都整合在了其解决方案之中。排序模型将数据中的实例转换成按序排列的连续值，进一步揭示了问题的本质。

与此同时，tuzzeg 那极具创新性的句子层次模型专门在训练数据中的单个句子实例上训练。在预测测试数据时，他将实例分割成句子，单独进行评估，然后只选取句子中的最高分数。在自然语言中，侮辱性言论往往集中在一段话中的某部分，tuzzeg 这样做就是为了适应这一点。

tuzzeg 的模型构造了超过 100 个特征组（一个基于词干的二元标记就是一个特征组，这里特征组指通过二元标记过程产生的特征向量），其中最重要的几个组（按影响排序）如下所示。

```
stem subsequence based         0.66
stem based (unigrams, bigrams) 0.18
char ngrams based (sentence)   0.07
char ngrams based              0.04
all syntax                     0.006
all language models            0.004
all mixed                      0.002
```


结果非常有趣，可以看出，一些现在并未使用的衍生特征对生成可用解决方案非常重要，尤其是基于序列的特征只是针对最初特征集的简单平移，因此易于添加额外特征。

```
def subseq2(n, xs):
    l = len(xs)
    return ['%s %s' % (xs[i], xs[j]) for i in xrange(l-1)
            for j in xrange(i+1, i+n+1) if j < l]

def getSubseq2(seqF, n):
    def f(row):
        seq = seqF(row)
        return set(seq + subseq2(n, seq))
    return f

Subseq2test = getSubseq2(line, 2)
```

该方法效果极佳。建议试用一下 tuzzeg 的解决方案，也可以看看该项目库中的 `score_trolls_withsubseq.py` 脚本，它整合了强大的额外特征。

添加额外特征后，AUC 提升显著（见图 6-4）。

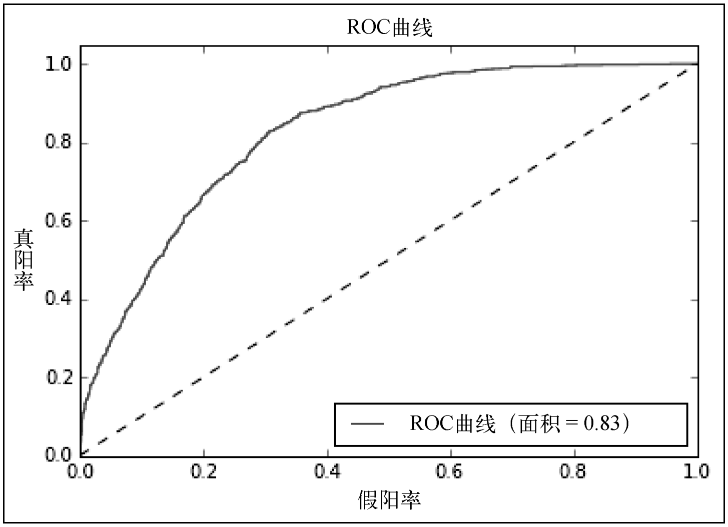


图　6-4

运行这段代码，AUC 达到了 0.834，结果很不错。这体现了考虑周全又充满创新的特征工程的强大。虽然本章生成的具体特征在其他场景中也很有用，但特定的假设（比如评论包含多个句子，充满敌意的言论只出现在特定几个句子中）能产生非常有效的特征。

本章基本都在用测试数据检验推断，然而现实中难有这样的条件。尽管我们没有投机取巧地私自查看测试数据，但了解排行榜上的得分情况有助于我们确立正确目标。第 8 章将用更严谨、更实际的方式应对另一个棘手的 Kaggle 问题，也将深入探讨集成模型。

6.3　扩展阅读

本章开头的引言来自 Kaggle 博客 No Free Hunch（值得阅读），详见 http://blog.kaggle.com/2014/08/01/learning-from-the-best/。

探讨自然语言处理任务的资源有很多，下面这篇文章谈到了 8 个方面，非常详尽：http://textminingonline.com/dive-into-nltk-part-i-getting-started-with-nltk。

如果想动手实践，不妨试试 Kaggle 的 Knowledge NLP 任务（https://www.kaggle.com/c/word2vec-nlp-tutorial/details/part-1-for-beginners-bag-of-words），该任务非常适合试验本章提及的技术。

本章提到的 Kaggle 大赛可参见 https://www.kaggle.com/c/detecting-insults-in-social-commentary。

要想深入了解 ROC 曲线和 AUC 测度，可以阅读 Tom Fawcett 的论文 “An introduction to ROC analysis”。

6.4 小结

本章介绍了许多实用且适用性强的技术，并通过一系列严谨的步骤，将凌乱且复杂的文本数据转化成了大量高效特征。首先应用了一些数据清洗技术，剔除了大量噪声和问题元素，然后用 POS 标记和词袋法将文本转化成了特征。其间介绍了一系列应用广泛且强大的技术，可解决许多自然语言处理的难题。

试验了多种单个模型和集成模型后可得知，当更智能的算法无法产生良好的结果时，彻底且富有创造性的特征工程可以显著提升模型性能。

第 7 章

特征工程 II

7.1 介绍

前面介绍了特征工程的重要性。上一章探讨了能高效将原始数据转化为特征并进行特征选择的技术，应用高级机器学习算法能高效处理这些特征。

混乱的输入会导致混乱的输出，这句箴言描述的情况在现实中时有发生。前几章讲过，图像识别和自然语言处理任务都要求数据经过充分处理。本章将接触一种更普遍的数据：从实际应用中收集的量化数据或类型数据。

本章将处理的数据在许多场景中都很常见，比如森林中的传感器、游戏机或金融交易捕获的遥测数据、煤矿地测信息，以及研究过程中收集的生物测定数据。不论怎样，核心原则和技术都是不变的。

本章将介绍如何审核并清洗数据、如何提升数据质量、如何将数据转换成适于机器学习的形式，以及如何创造性地增强数据。

总体而言，本章将讨论如下概念。

- 创建特征集的不同方法，以及特征工程的局限性。
- 如何用各种技术增强并提升初始数据集。
- 如何运用领域知识来选择有效措施，以转换现有数据并提高其整洁度。
- 如何测试单个特征值和特征组合值，只保留需要的特征。

首先详述基本概念，本章末尾将进行多次迭代试验，并通过专门的测试来展示所构建特征的益处。

7.2 创建特征集

事关机器学习成功的最重要因素是输入数据的质量。如果数据引人误解、标准化不当或信息

量不足，模型再好也无法达到在准备充分的数据上运行的效果。

有时可以使用特定数据集，或者访问有用而多样化的大型数据源。然后恰当运用知识和技能，用这些数据构造出有用的特征集。

总的说来，掌握如何构建良好的特征集非常有用，保证了我们可以审核和评估任何新数据集。本章将介绍一种设计过程和一套技术，展示如何轻松地创建有用的特征集。

因此，下面首先探讨一些常用技术，可用这些技术扩展或转换现有特征，以及为模型创建大量有用的参数。

如前所述，特征工程技术的高效使用存在限制，必须留意经过特征工程处理的数据集隐含的风险。

7.2.1 为机器学习应用构建特征

前面讲过如何处理数据质量问题，以及如何创造性地借助已有数据的维度连接外部数据。

充分理解了手头数据并检查过其质量后，还需要做大量工作，以便后续根据数据构建有效模型。

1. 用标准化技术提升特征的可学习性

直接将未处理过的数据输入机器学习模型的主要问题是，算法对不同变量的相对大小非常敏感。如果数据集中存在值域不同的多个参数，有些算法会认为方差更大的变量会比值更小、方差更小的变量更有意义。

解决该潜在问题的关键是**再标准化**（也称“再缩放”），即调整参数值的相对大小，同时保持每个参数值的初始排序（即单调转化）。

如果训练前就对输入数据进行标准化，可以考虑使用梯度下降算法（绝大多数深度学习算法会涉及）。下面通过一些图说明其原因。给定的一系列训练步骤如图 7-1 所示。

7

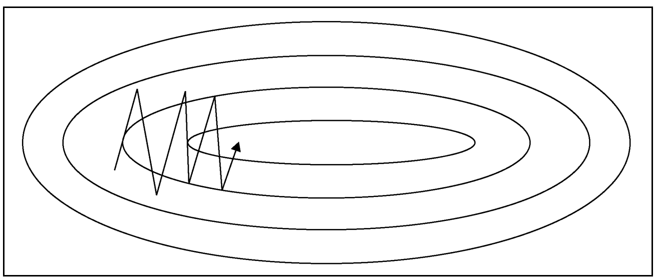


图 7-1

应用于未经标准化的数据时，这些训练步骤可能不会有效收敛（如图 7-2 中左边的例子所示）。

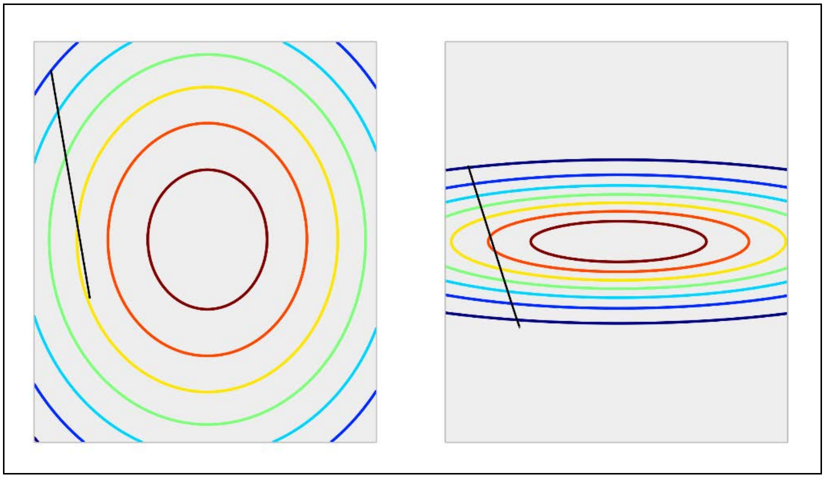


图　7-2

如果每个参数的取值范围都不同，那么训练模型的参数空间会高度扭曲且极其复杂。该空间越复杂，在其中训练模型的难度也就越大。这可以用一个比喻来描述（对此的完整解释，参见 7.4 节的参考说明）。不妨将训练过程中的梯度下降模型想象成大理石从斜坡上滚下。这些大理石可能会在斜坡上（即模型的目标函数所创建的表面，其中目标函数指学习函数，模型要训练将其输出最小化）的鞍点或其他复杂几何结构处**卡住**。但对标准化后的数据而言，该平面的形状较为规则，训练过程也就更高效。

典型示例是 0~1 的线性再标准化，通过该方法，参数的最大值将再标准化为 1，最小值为 0，中间值则落在 0~1，具体指其原始值相对于最值（最大值和最小值）的比例。按照该转化规则，向量[0, 10, 25, 20, 18]将变为[0, 0.4, 1, 0.8, 0.72]。

这种转化的特别之处是，再标准化后，原始形式下不同大小的多个数据点将落在同一个值域范围内，以便机器学习算法在更有意义的信息上进行训练。

这是最简单的再标准化方法，一些非线性标准化方法在适用场景中会效果更佳，比如平方标准化、平方根标准化以及（可能最常用的）对数标准化。

参数值的对数标准化在物理学或其他经常受到幂次法则（比如 x 的线性增长会引起 y 的指数增长）影响的数据中很常见。

与线性再标准化不同，对数标准化调整的是数据实例的相对间距。这是一把双刃剑：一方面，对数标准化能很好地处理离群点。例如图 7-3 中的概括统计量描述了一个虚构群体中的个人净资产。

统　计	个人资产
最小值	1
下四分位数	42.5
均值	3205433.343
中位数	600
上四分位数	1358
最大值	10000000000

图　7-3

进行再标准化前，群体分布严重偏向净资产值异常的个体。每一等分的实例分布如图 7-4 所示。

范　围	计　数
0 > 0.1	3060
0.1 > 0.2	0
0.2 > 0.3	0
0.3 > 0.4	0
0.4 > 0.5	0
0.5 > 0.6	0
0.6 > 0.7	0
0.7 > 0.8	0
0.8 > 0.9	0
0.9 > 1	1

图　7-4

进行对数标准化后，分布看起来合理多了（见图 7-5）。

范　围	计　数
0 > 0.1	740
0.1 > 0.2	1633
0.2 > 0.3	544
0.3 > 0.4	141
0.4 > 0.5	0
0.5 > 0.6	1
0.6 > 0.7	0
0.7 > 0.8	1
0.8 > 0.9	0
0.9 > 1	1

图　7-5

可以进一步执行标准化，从而更巧妙地拉伸该分布的上半部分。在本例中，以 10 为底的对数标准化能够极大地降低这些离群值的影响，既保留了这些离群点，又不会丢失底部的细节。

如上所述，需要注意，在某些情况下，比较接近的实例之间也会得到这种增强，这可能会使一些参数值的噪声增大，引起误导——这些值之间差距较大。这种做法不会导致对数标准化对离群点的处理产生负面影响，但会在初始值非常接近且值比较小的实例之间产生负面影响。

通过对数标准化引入非线性而产生的问题需要特别注意，而且通常非线性标准化只适用于我们了解且存在非线性关系或趋势的变量。

2. 构造高效的衍生变量

再标准化是许多机器学习应用（比如大多数神经网络）的预处理环节的常规部分。除了再标准化，还有其他许多数据准备技术能够有效地减少模型内的输入参数，提升模型性能。其中最常用的就是衍生度量，该技术能综合多个已有数据点并将其表示为简单测度。

这类衍生测度十分常见，比如加速度（速度随时间变动的函数）、身体质量指数（身高、体重和年龄的函数）和用于评测股票的**市盈率**（price-earnings ratio，P/E）。本质上来说，几乎所有衍生分数或复杂测度都是多种成分构成的联合得分。

比较常见的数据集往往有很多知名的测度。即使在相对知名的领域，将已有数据与领域知识相结合都会得到一些非常有效的衍生测度和数据转换技巧。衍生方法的一些相关概念如下所示。

- **两个变量的组合**：对参数 n 进行乘除或标准化，并作为参数 m 的函数。
- **随时间变化的度量**：典型的度量示例是加速度或 7D 变化。更复杂的情况下，比起直接处理当前值和过去值，潜在的时间序列函数的斜率是更有用的参数。
- **基线的减法**：用基准预期（一种直接预期，比如**基线流失率**）按照该基线重新调整参数，这能够令同一个变量暴露出更多信息。在用户流失案例中，实际与预期的偏差可以用一个参数来描述。同样，在股票交易中，可以根据开盘价研究收盘价。
- **标准化**：如上例所示，有的参数值是根据其他参数值或基线来进行标准化的，因此可以根据其他变量的基本性质进行动态计算。例如**交易失败率**，除了可以视作原始（或再标准化）统计，往往还需要根据未完成交易量进行标准化。

以上不同元素的创造性重组可以产生非常好的效果。例如有时描述一段时间内客户互动斜率（下降或上升）的参数必须以该顾客是否曾经频繁互动或极少互动为条件，因为在不同背景下，互动轻微下降的含义可能不同。高效地、创造性地构造特征集来捕捉给定领域内的细微之处，是数据科学家的职责所在。

前面仅探讨了数值型数据，而有用的数据往往是非数值型的，比如编号或类型数据。因此，下面将介绍一些将非数值型特征高效转化为可用参数的技术。

3. 转换非数值型特征

非数值型特征的处理是一个普遍问题，该问题有时很棘手，并且往往没有通用的解决方案。有价值的信息往往编码在非数值型的简写值中。例如在股票交易中，股票本身（如 AAPL）和买卖双方的身份都是我们感兴趣且与问题息息相关的信息。更进一步，我们可能希望一些股票的交易和其他股票出现差异，哪怕是在同一行业，企业内部的组织设置上的差异（这种情况很常见）也会包含重要信息。

很多时候，构建一个或多个聚合度量是常用方法。如前所述，一个典型示例是基于可能的发生次数构建衍生测度（比如两个时间窗口之间的次数变化）。

构建概括统计量或减少数据集的行数，会降低模型本可以学习到的信息量，从而带来风险（可能导致模型更加脆弱，并引起过度拟合）。因此，广泛聚合和减少输入数据往往不可取。这一点对深度学习技术而言更是如此，比如第 2 章到第 4 章探讨和使用的算法。

因此，不要广泛使用基于聚合的方法，而要选择将字符型数据转化为数值型数据的方法。另一类常用的技术是编码，其中最常用的编码方法是独热编码，即把类型响应变量（比如年龄组别）转化成二元变量，其中每种响应选项（如 18~30 岁）都由其独有的二元变量表示。图 7-6 展现得更直观。

案例	年龄	性别
1	22	男性
2	25	男性
3	34	女性
4	23	男性
5	25	女性
6	41	女性

图 7-6

编码后，由类别型变量和连续型变量构成的数据集就变成了二元变量的张量（见图 7-7）。

案例	年龄_22	年龄_23	年龄_25	年龄_34	年龄_41	女性	男性
1	1	0	0	0	0	0	1
2	0	0	1	0	0	0	1
3	0	0	0	1	0	1	0
4	0	1	0	0	0	0	1
5	0	0	0	0	0	1	0
6	0	0	0	0	1	1	0

图 7-7

这种表示法优势明显，无须聚合或冒着数据信息量减少的风险，就挖掘到数据所包含的重要标签信息。此外，独热编码还能将编码后变量的特定编号分割成独立的特征，即可以或多或少地识别出特定变量的有意义代码，而只保留重要值。

另一种主要用于文本代码的有效技术是**散列法**。简而言之，散列是将数据转化成数值表示的函数。很多人熟悉这个概念，因为它常用于编码敏感参数，以及概括庞大的数据。但为了最大限度地利用散列法，必须理解该方法的原理和作用。

散列法可以将文本短语转化成数值，并作为该短语的标识符。虽然散列法类型多样，应用广泛，但在文本特征处理领域，简单的散列法也能直截了当地将关键字和编号转化成数值参数，方便高效建模。

一种简单的散列法可以将每个字母字符转换成对应的数字。a 变成 1、b 变成 2，以此类推。对单词或短语对应的值求和可以生成散列值。例如短语 cat gifts 就可以按照以下方法进行转换。

```
Cat: 3 + 1 + 20
Gifs: 7 + 9 + 6 + 19
Total: 65
```

该散列法非常糟糕，原因有二（先不考虑输入数据中包含垃圾词的情况）。其一，没有限制输出数。散列法的关键是降维，因此必须对其可能的输出数加以限制。大部分散列法会限制其输出的数字范围，因此选择散列法时需要考虑模型包含的特征数量。

> 通常的做法是将 2 的幂次作为散列法范围，在散列过程中逐位进行运算来加速。

其二，这种散列法对单词转换的影响不大，反而很小。如果将 cat 变成 bat，那么散列法的输出也应该明显改变，而实际上只是数值变化了 1 而已（变成了 64）。总体而言，好的散列函数应该在输入文本出现极小改变时，输出发生较大变化。部分原因是，语言结构非常一致（因此分数也会接近），但给定结构内略有不同的名词和动词组合会呈现截然不同的含义。例如 the cat sat on the mat（猫坐在垫子上）和 the car sat on the cat（车在猫上）意思就很不同。

介绍过了散列法，下面详细介绍散列的一些技巧。如果将每个词都转化成散列数值代码，就会出现大量的**散列碰撞**，即两个词的散列值相同。这当然是很糟糕的。

Zipf 分布能描述不同词语的词频分布。它将频率排第 n 名的词语的概率近似地表示为 $P(n) = 0.1/n$，其中 n 最大不超过 1000（Zipf 定律）。这确保了每个词比前一个词频率更低。当 n 大于 1000，词语就会变得十分模糊，这时数据集中很难存在两个散列值相同的词。

此外，好的散列函数会限制值域，而且极易受到输入数据微小变化的影响。这些性质令散列碰撞在很大程度上与词语使用频率相分离。

Zipf定律和好的散列函数（即散列碰撞概率与词语使用频率相分离的散列函数）的概念意味着散列碰撞的概率极小，而当发生散列碰撞时，这两个词都极有可能是频繁使用的词。

这赋予了散列法一个特殊性质。与不使用散列法的词袋特征进行的训练相比，散列法能在不降低模型性能的基础上，更大幅度地降低文本输入数据的维度（从成千上万个自然出现的词语减少到几百个，甚至更少）。

恰当应用散列法，可以实现许多妙用，其中包括增强之前讲过的技术（特别是词袋法）。不同散列法的参考资料参见 7.4 节。

7.2.2 运用特征选择技术

前面介绍了构造特征的许多技术，以及创造性的特征工程的潜力，下面可以把已有特征改造成更有效的变体了。运用这些特征工程技巧时，可能会因为创建过多衍生特征而变得难以管理。

不加限制地添加特征会令模型变得脆弱，还会令某些模型出现过度拟合。这与想模拟的趋势的复杂程度密切相关。最简单的情况是，想识别两大分组之间的显著区别，但随着需要拟合的模型将分类规则搞得愈发复杂，且两个组的规模越来越小，添加越来越多的特征会影响模型分类的一致性和高效性。

这是一个复合问题，因为不会每次都知道哪个参数或变化最适合具体问题。这种适用性因模型而异，比如决策森林使用单调变换（即保留数据原始顺序的变换，例如对数标准化）的效果就不如用未经标准化的原始数据的效果，但对其他算法而言，再标准化可能是非常有效的方法。

传统上，特征数量和参数数量都取决于构造函数的需求，该函数能将关键输入和理想的输出评估分数挂钩。在这种情况下，额外的参数需要以移动变量或干扰变量的形式整合进来。

每个新参数都会引入一个新维度，这使得模拟的关系更加复杂，也使得模型更容易过度拟合数据。例如引入一个相当于每个实例唯一标签的参数，那么算法只会学习这些标签，而根本无法预测新数据集。

微不足道的实例也可能产生严重问题，当特征将实例分割成非常小的组时，实例和特征的比例就变得举足轻重。简而言之，模型函数复杂度的提升会导致模型更容易出现过度拟合，而特征数量的增加更加重了这种影响。根据该原则，应该先从小数据集开始，并且只在确保参数能够提升模型的情况下添加参数。

然而，近来一种对立的方法论逐渐发展起来，并成为了**从事数据科学**的基本理论。它表明，可以构建大型特征集来包含所有具有潜在价值的特征，同时**减少**功能冗余的多余特征集。

在大型特征集（可能包含上百或上千种特征）上做决策的技术支持该方法论，其运行过程通常比较**粗暴**。这些技术会详尽地测试特征组合，连续或并行地运行模型，直到找到最有效的参数子集。

这些技术非常有效，这正是这种方法论流行的原因。了解这些技术很有必要，本章稍后将介绍这些技术的应用。

在特征选择中运用“暴力”技术的主要缺点是，我们很容易相信算法的输出结果，不管其选出的特征究竟意味着什么。应该在高效、无须领域知识的黑盒算法和理解所处理的问题之间找寻平衡。因此，本章将基于**扩张**和**缩减**这两种范式来运用技术，涵盖了不同的场景。首先介绍如何缩小手头的特征集，从庞大的特征中提取最有价值的特征。

执行特征选择

构建好大型数据集后，通常会面临一个问题：如何缩小选择范围，只保留最有效的数据。下面介绍一些支持特征选择的技术，有的技术本身起作用，有的则是一些算法的封装器。

这些技术包括相关性分析、正则化技术和**回归特征消除**（也称“递归特征消除”）。这些技术有助于筛选特征集，在每次处理新数据集时，它们还能省下大量人力。

- **相关性**

首先探究回归模型出现问题的主要原因：多重共线性，以此展开对特征选择的讨论。多重共线性指的是数据集中的特征存在较高的的相关性。例如比萨张数和比萨价格就存在共线性。

多重共线性分为两种：结构性的和基于数据的。结构性多重共线性出现在创建新特征时，比如根据特征 f 创建特征 $f1$，创建出的多个特征可能高度相关。基于数据的多重共线性则出现在两个变量受到同一诱因影响时。

两种多重共线性都会产生负面影响。具体而言，模型性能会受到所使用的特征组合的影响。使用共线特征时，模型性能就会下降。

处理这些情况的方法很简单：检验多重共线性是否存在，然后删除表现不佳的特征。表现不佳的特征通常指的是几乎无法提升模型性能的特征。它们之所以表现不佳，是因为复制了其他特征的信息，或单纯没有提供对所处理的问题有意义的数据。

还有一种特定多重共线性检验值得考虑，即检查数据相关性矩阵的特征值。特征向量和特征值是矩阵理论的基本概念，具有许多突出应用。更多细节见本章末尾。这里只需要知道，根据数据集生成的相关性矩阵的特征值是多重共线性的一种量化度量。特征值集合能够表明特征给数据集带来了多少“新信息”，较低的特征值表示数据可能与其他特征相关。例如以下代码构造了一个特征集，并为特征 0、特征 2 和特征 4 添加了共线性。

```
import numpy as np

x = np.random.randn(100, 5)
noise = np.random.randn(100)
x[:,4] = 2 * x[:,0] + 3 * x[:,2] + .5 * noise
```

在生成相关性矩阵并计算特征值时，我们发现：

```
corr = np.corrcoef(x, rowvar=0)
w, v = np.linalg.eig(corr)

print('eigenvalues of features in the dataset x')
print(w)

eigenvalues of features in the dataset x
[ 0.00716428  1.94474029  1.30385565  0.74699492  0.99724486]
```

显然，第 0 个特征[①]很可疑。下面调用v来检查该特征的特征值。

```
print('eigenvalues of eigenvector 0')
print(v[:,0])

eigenvalues of eigenvector 0
[-0.35663659 -0.00853105 -0.62463305  0.00959048  0.69460718]
```

特征 1 和特征 3 值较小，由此可以推断特征 2 和特征 4 与特征 0 具有较高的多重共线性。应该在进行下一步前先移除其中两个特征。

- **LASSO**

正则化方法是最有效的特征选择技术，因为它能提供稀疏解，即把较弱的特征归零，只留下一部分特征及其真正有效的系数值。

最常用的正则化模型是 L1 正则化和 L2 正则化，两者在线性回归领域分别称为“LASSO”和“岭回归”。

正则化方法的原理是给损失函数添加惩罚项。不是将损失函数 $E(X,Y)$最小化，而是添加了一个项 $a||w||$，其中超参数 a 与正则化的量有关（帮助调整正则化强度，从而调整挑选出的原始特征集比例）。

LASSO 正则化使用的特定惩罚函数是 $a\sum ni=1|wi|$。每个非零系数都会增大惩罚项，迫使较弱特征的系数归零。scikit-learn 为超参数提供的参数优化组件可用于选择合适的惩罚项。本例将用`estimator.get_params()`执行网格搜索，以选择合适的超参数值。有关网格搜索操作的更多信息，参见 7.4 节。

scikit-learn 的逻辑回归带有用于分类的 L1 惩罚项。LASSO 模块也提供了线性回归功能。下面将 LASSO 应用于一个示例数据集，即波士顿住房数据集。

```
fromsklearn.linear_model import Lasso
fromsklearn.preprocessing import StandardScaler
fromsklearn.datasets import load_boston
```

① Python 中的索引从 0 开始。——译者注

```
boston = load_boston()
scaler = StandardScaler()
X = scaler.fit_transform(boston["data"])
Y = boston["target"]
names = boston["feature_names"]

lasso = Lasso(alpha=.3)
lasso.fit(X, Y)

print "Lasso model: ", pretty_print_linear(lasso.coef_, names, sort = True)

Lasso model: -3.707 * LSTAT + 2.992 * RM + -1.757 * PTRATIO + -1.081
* DIS + -0.7 * NOX + 0.631 * B + 0.54 * CHAS + -0.236 * CRIM + 0.081 *
ZN + -0.0 * INDUS + -0.0 * AGE + 0.0 * RAD + -0.0 * TAX
```

原始数据集的多个特征系数都归零了，相关性的提升令解愈发稀疏。例如 `alpha = 0.4` 时结果如下所示。

```
Lasso model: -3.707 * LSTAT + 2.992 * RM + -1.757 * PTRATIO + -1.081
* DIS + -0.7 * NOX + 0.631 * B + 0.54 * CHAS + -0.236 * CRIM + 0.081 *
ZN + -0.0 * INDUS + -0.0 * AGE + 0.0 * RAD + -0.0 * TAX
```

特征选择技术 L1 正则化的值可以立即得出。然而请注意，L1 正则化回归并不稳定。当数据内的特征相关时，即使数据变化很小，也可能令系数显著改变。

L2 正则化（或岭回归）以另一种方式处理特征系数，可以有效地解决该问题。L2 正则化在损失函数中加入一个额外的惩罚项，即 L2 范数惩罚项，其形式为 $a\sum ni=1w\text{^}2i$。显然，与 L1 惩罚项（$a\sum ni=1|wi|$）不同，L2 惩罚项使用了平方后的系数。这导致系数值分布更加平均，且相关特征会接收相似的系数值。这可以提高稳定性，系数不会因为数据的微小变化而显著波动。

然而，L2 正则化并不像 L1 正则化那样直接有助于特征选择。但因为有趣的特征（具有预测能力的特征）的系数通常非零，所以作为探索工具的 L2 正则化更有用。它有助于我们推断分类中特征的质量，还比 L1 正则化更加稳定可靠。

- **回归特征消除**

回归特征消除是一种贪婪迭代算法，可用作其他模型（比如支持向量机）的封装器，即在输入数据的不同子集上反复运行。

应用回归特征消除的目标与 LASSO 和岭回归相同，即找到表现最好的特征子集。正如其名，该技术每次迭代时剔除一个特征，用剩余特征重复该过程，直到数据集内的所有特征都消除为止。特征消除的顺序就是其排名。随着迭代次数增加，子集会越来越小。多次迭代之后，每个特征都会得到精确的评分，并且相关子集会被挑选出来以备后用。

下面举例说明该算法的原理，将用到之前讲过的数字数据集。

```
print(__doc__)

from sklearn.svm import SVC
fromsklearn.datasets import load_digits
fromsklearn.feature_selection import RFE
importmatplotlib.pyplot as plt

digits = load_digits()
X = digits.images.reshape((len(digits.images), -1))
y = digits.target
```

我们将通过**支持向量分类机**的SVC运算符将支持向量机作为基础评估器，然后对模型应用回归特征消除封装器。回归特征消除接收多个参数。第一个参数用于引入所选择的评估器。第二个参数是n_features_to_select[①]。有时特征集会包含很多相互关联的特征，而这些特征子集所包含的多元分布可能会对分类有重要影响，此时可以考虑将这两个或多个特征进行组合。

可以通过设置 step 在每次迭代时删除多个特征。比如给定一个 0.0~1.0 的值，那么每一步都会删除特征集中的一部分，这部分的占比就是 step 的参数值。

```
svc = SVC(kernel="linear", C=1)
rfe = RFE(estimator=svc, n_features_to_select=1, step=1)
rfe.fit(X, y)
ranking = rfe.ranking_.reshape(digits.images[0].shape)

plt.matshow(ranking)
plt.colorbar()
plt.title("Ranking of pixels with RFE")
plt.show()
```

如前所述，该数字数据集中的每个实例都是 8×8 的手写数字图像，如图 7-8 所示。每个图像都位于 8×8 网格的中部。

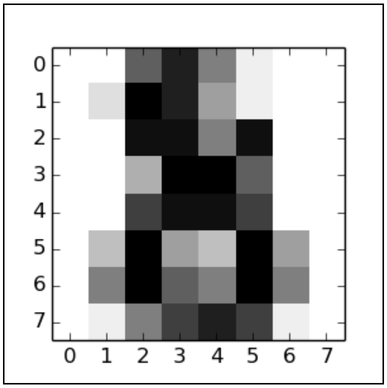


图 7-8

① 用于设定特征筛选数。——译者注

当在数字数据集上应用回归特征消除时，可以看到该技术在排序时明显捕获了这些信息（见图 7-9）。

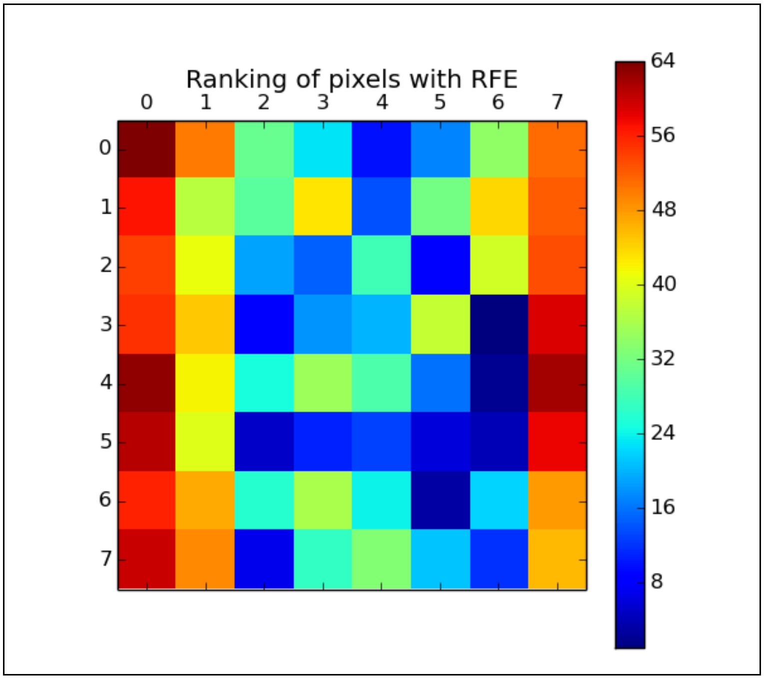


图　7-9

首先被删除的像素在图像（通常为空）垂直边缘的内侧附近。然后，算法一般会删除垂直边缘附近或接近图像顶部的空白区域。保留最久的像素是那些最能区分不同数字的像素，即在一些数字中出现，而不在其他数字中出现的像素。

本例有助于阐释回归特征消除的原理，但并未展现这项技术的一致性如何。回归特征消除的稳定性取决于基础模型的稳定性，在某些情况下，岭回归能得到更稳定的解。（有关实例和条件的更多信息，参见 7.4 节。）

- **遗传模型**

前面提过，一些算法有助于我们在庞大的参数集上执行特征选择。其中最好的技术是遗传算法，它能模拟自然选择，以生成越来越有效的模型。

用于特征选择的遗传算法方案大致如下。

- ❑ 一系列初始变量（“预测器”一词仅用于该语境下）组合成多个子集（候选），并对每个候选计算性能测度。

- ❑ 候选中表现最佳的预测器将再随机组合，进入模型的下一次迭代，又称“下一代”。
- ❑ 经过这次再组合后，每个子集都有可能变异，因而需要从中添加或删除预测器。

这种算法通常会迭代多次。最适宜的迭代数取决于数据集和模型的复杂程度。遗传算法中的性能和迭代数的关系与梯度下降技术相似，随着迭代数增加，性能提升会非线性地下降，最终在过度拟合风险上升前达到最小值。

为了找到最高效的迭代数，可以用训练数据进行测试。通过多次运行模型，并绘制**均方根误差**（root mean squared error，RMSE）曲线，可以找到输入数据和模型配置对应的最佳迭代数。

下面详述每次迭代的细节。特别是如何创建候选子集、如何评估效果，以及如何进行再组合。

候选项最初配置为可用预测器的随机样本。第一代所使用的预测器数量没有硬性规定，取决于可用的特征数，但第一代候选的样本数通常为可用特征数的 50%~80%（特征越多，比例越小）。

拟合度指标可能很难定义，通常会使用两种形式的交叉验证。内部交叉验证（在不比较模型的情况下，仅在其自身参数的环境中测试每个模型）通常用于追踪给定迭代的效果。内部交叉验证中的拟合度指标则用于选择下一代再组合用的模型。此外需要外部交叉验证（测试未在任何迭代中用于验证的数据集），以确保查询过程生成的模型不会对内部训练数据过度拟合。

再组合受限于 3 个关键参数：变异概率、交叉概率和精英机制。其中最后一个为可选参数，可用于保留目前一代中表现最佳的模型中的前 n 个，以避免再组合过程中完全丢失高效的候选。这些候选也将用作变体或下一代候选的父类。

变异概率定义了下一代模型随机再调整（通过添加或删除预测器，一般为 1 个）的概率。变异能帮助遗传算法维持候选变量的广泛覆盖，降低解受限于局部空间的风险。

交叉概率则定义了一对被选中在下一代模型中进行再组合的概率。交叉算法有多种：每个父类的特征集的一部分可能要相互结合（比如第 1 个父类的一半加上第 2 个父类的一半）为子代，或随机选择所使用的每个父类的特征。默认使用两个父类的共同特征。在两个父类的独有预测子集合中随机采样是一种常用方法。

以上是常规遗传算法的主要部分，该算法可用作现有模型（比如逻辑回归、支持向量机等）的封装器。这里提到的技术有许多变体，并且不同量化领域的特征选择技术也存在些许区别。下面将理论知识应用于一个实际案例。

7.3 特征工程实战

根据建模方法的不同，特征工程的实际效果也会有所差异。比如相较于一些较浅的模型，深度学习算法仅需较少的特征工程就能有更好的表现，而且改进结果所需要的工作量也较少。

理解需求的关键是，快速、深入地理解从数据获取到建模的全过程。首先要设定目标，以明确模型所需的准确度，然后找到一个可接受的最快的实现方案并执行。尽可能学习结果所透露出的有效信息，并计划下一次迭代。

下面在实践中演示该过程——迭代式地不断建模并逐步提高模型质量，以处理一个没见过的高维数据集。

我最近住在温哥华。这座城市有许多优点，但最大的缺点是，城市的交通让人难以捉摸。无论是开车还是搭乘 Translink 的“天空列车”系统（一种单轨的过山车式高速铁路），都很容易遇到难以预测的延迟和堵车问题。

本着将新的特征工程技能付诸实践的精神，下面考虑能否提升这座城市的交通体验。步骤如下所示。

- 通过编程获取多个 API 的数据，其中包括文本流数据和天气数据。
- 用特征工程技术从原始数据中构造衍生变量。
- 通过计算通勤延迟风险评分来测试特征集。

本例不关注如何构建高性能模型，而关注如何构造自洽的解，即可以根据所处地区进行调整和应用的解。虽然该方法符合本章的目的，但有另外两个重要原因。

首先，分享和利用 Twitter 数据时会出现一些问题。Twitter API 的部分使用条款需要开发者保证，任何对时间轴或数据集的状态调整（比如删除一条推文）都必须能通过从 Twitter 获取或公开分享的数据集复现。这一点导致本章的 GitHub 库很难包含真实的 Twitter 数据，因为用户需要自己构建数据流并累积数据，这提高了最后根据数据流构建的下游模型复现结果的难度，而且数据的其他变化（比如季节性变化）也可能会影响模型效果。

另一个原因非常简单：并非所有人都住在温哥华。为了向最终用户提供价值，需要考虑一种可调整的泛化解，而不是限于地理位置的特定解。

因此，稍后展示的代码将用于构建或开发。这些代码可作为成功的商业应用的基础，或单纯作为一种有用的由数据驱动的生活技巧。在此前提下，阅读本章内容（并利用相关目录的代码），然后创造出适合具体情况、可获得本地数据，并能满足个人需求的新应用。

通过RESTful API获取数据

首先需要收集一些数据。这里要找的是频率足够高（最好每个通勤时段都至少有一条记录）的大量时间戳数据，以进行模型训练。

Twitter API 是不错的选择，因为易于获取最近的推文数据。具体有两种方法。

首先，可以收集官方高速运输管理局（特别是有关公共汽车和火车的企业）的推文。这些企业能提供延误和服务中断的运输服务信息。这些信息格式一致，有助于给数据添加标签。

此外，可以搜索感兴趣的区域的相关推文，用自定义词典捕捉与服务中断事故或其原因相关的词语，从而了解通勤者的情绪。

除了利用 Twitter API 来获取支持模型的数据外，还可以利用其他 API 提取大量信息。其中尤为重要的数据源是**必应交通 API**，它能够提供特定用户所处区域交通堵塞或中断事故的信息，且易于调用。

此外，还可以利用**雅虎天气 API** 的数据。该 API 能根据邮政编码或位置输入给出特定地点的当前天气状况。它所提供的当地气象信息非常丰富，包括但不限于温度、风速、湿度、气压和能见度。此外，它还提供当前天气状况的文字描述以及天气预报。

尽管可以加入分析的数据源还有很多，首先看看如何处理这份数据。

1. 测试模型性能

需要做的是评估当前每天通勤中断的风险，最好能预知通勤风险，从而采取行动（比如早点出门）来降低风险。

对此，需要以下 3 点。

- 了解模型输出；
- 用于量化模型性能的方法；
- 能够根据方法为模型性能评分的目标数据。

就这几点的重要性，可以展开讨论。实际上，一些模型是有目的的信息汇总。通勤风险分就很有用，因为它能生成一些新信息。

实际情况是，必须要有一个性能测度标准。该标准可以单纯是对模型输出结果的满意程度，但这种标准是很有用的。因此，性能测度的量化非常重要，即使该模型有自己的测度方式（甚至是无须测度的无监督学习）。性能测试不可忽略，至少需要一个量化的性能测度来迭代式地优化模型。

比较明智的做法是，设定模型的输出是对某日的上班通勤（从家到公司）状况的数值评分（评分范围为 0~1）。该评分的计算有几种方法，其中最常用的方法可能是对数据应用对数标准化。对数标准化有很多优点，而且对于本案例来说是比较好的做法（毕竟通勤延迟时间的分布很可能遵循幂次法则）。目前不对这些得分执行数据转换，而是等模型输出结果后再进行检验。

0~1 的得分很难给出切实可行的指导意见。因此，可能要用时段系统（比如高峰、中峰和低峰），时段间的划分边界会限制在 0~1。简而言之，需要将问题当作具备类型输出（类标签）的多分类问题来处理，而不是连续数值型输出的回归问题。

这样能够提升模型性能。（确切地说，自由误差限度会放宽到相关时段的对应宽松度，而这是一种非常宽松的性能测度。）同样，第一次迭代就引入这种变化并不合适。评估真正通勤延迟的分布之前，无法知道在哪里绘制类边界。

接下来需要考虑如何度量模型的性能。恰当的评分方法通常取决于问题的特征。对分类器性能进行评分的方法有很多。（关于机器学习算法性能度量的更多方法，参见 7.4 节。）

决定性能测度是否合理的方法之一是“混淆矩阵”。混淆矩阵是一张可能事件的表格，通常能在统计建模中描述预测标签和实际标签的差异。输出训练后模型的混淆矩阵（尤其是类更多的多类问题）是一种常见做法，因为混淆矩阵可以得出分类失败的类型的重要信息。

由此可知，采用混淆矩阵可以更直观地展现模型的性能。不妨查看以下的简化矩阵（见图 7-10），看看其中是否存在未关注到的偶发事件。

		实际结果	
		真	假
预测	真	真阳性	假阴性
	假	假阳性	真阴性

图　7-10

本例需要考虑所有 4 种可能事件。假阴性指会遇到意料之外的延迟，而假阳性指不必提前出门。这意味着，需要一个既有高敏感度（真阳性），又有高特异性（真阴性）的性能测度。因此，最理想的选择就是 AUC。

第二个大问题是如何计算该分数。需要预测变量的目标值。好在这些目标值容易获取，毕竟我每天也需要上下班。只需要一只秒表，在固定的时间开始计时，经过固定的通勤路线，就能记录每天的通勤用时。

需要注意的是，该方法有很大的局限性。作为数据源，我受限于自己的生活习惯，比如早上需要喝咖啡来提神。同理，我自己的固定通勤路线可能有着其他路线所没有的本地趋势。如果能收集到许多人和许多路线的通勤数据，就会好很多。

但从某种程度上说，现在的目标数据还不错。因为我想对自己通勤路线上的中断事故进行分类，而不希望通勤时间的自然变动在训练过程中被误解，比如受到其他通勤者或通勤路线的影响。此外，性能优越的模型应该会忽略我们预料到的日常自然变化。

此外，很难判断模型性能是否已经足够好。确切地说，很难知晓模型的表现何时会超过预期。况且我无法精确预测通勤延迟，个人的预测似乎也无法泛化到其他地区的其他通勤者。为模型训练设定相当主观的目标似乎并不明智。

不妨换种思路，尝试让模型超过一个极低的门槛，即简单地假设每天都不会出现通勤延迟。该目标有着相当讨喜的性质，它能体现出我们对现实生活的美好祈愿（可以每天不用早起，不用考虑交通中断的影响）。

在 85 个目标数据实例中，观测到了 14 次通勤延迟。基于这份目标数据和创建的评分测度，要超过的最低阈值应为 0.5。

2. Twitter

由于案例分析是基于温哥华这座城市进行的，因此可以研究第二种 Twitter 数据源，具体而言，就是温哥华公共交通管理局 Translink 的官方服务声明。

- **Translink Twitter**

前面提到，数据已经具备良好的结构，而且有助于文本挖掘和后续分析。用前两章介绍过的技术处理数据后，就能清洗文本并将其编码成有用的特征了。

下面将使用 Twitter API 获取一段时间内 Translink 的推文。Twitter API 易于使用，可以通过 Python 轻松调用。（如何使用 Twitter API 的详细指导，参见 7.4 节。）本例需要从推文中提取日期和正文文本，其中后者几乎包含了需要了解的所有信息，具体如下所示。

- 推文的性质（是否延迟）
- 影响的站点
- 有关延迟性质的一些信息

复杂之处是，Translink 账户会同时发布“天空列车”线路和公共汽车线路的服务中断信息。好在该账户描述问题的用词（比如服务类型或服务对象）非常统一，并且会用特殊标签（例如公共汽车路线信息用#RiderAlert，火车相关信息用#SkyTrain，两种服务的一般警报用#TransitAlert，比如法定节假日）来区分服务中断的对象。

同理，可以预期延迟事件会用“延迟”（delay）一词来描述，绕路事件会用“绕路”（detour）一词来描述，分流事件也会用“分流”（diversion）一词来描述。这意味着，可以用关键词筛掉不想要的推文。感谢 Translink！

> 本章使用的数据见本章附带的 GitHub 仓库，文件为 translink_tweet_data.json。下载脚本也含在本章代码中。要想利用这些资源，需要向 Twitter 申请一个开发者账户。申请流程很简单，有相关文档说明，也可以在那里注册。

获取推文数据后，下一步是清洗和规范正文文本。根据第 6 章所讲，在输入数据上运行 `BeautifulSoup` 和 `NLTK`。

```
from bs4 import BeautifulSoup

tweets = BeautifulSoup(train["TranslinkTweets.text"])

tweettext = tweets.get_text()

brown_a = nltk.corpus.brown.tagged_sents(categories= 'a')

tagger = None
for n in range(1,4):
    tagger = NgramTagger(n, brown_a, backoff = tagger)

taggedtweettext = tagger.tag(tweettext)
```

清洗工作的强度不必像上一章清洗喷子数据集那样高。Translink 的推文高度格式化，不含非 ASCII 码字符或表情符号，因此无须使用第 6 章必须使用的特定“深度清洗”正则化脚本。

Translink 的推文集是全部小写、规范化且用词符合词典规范的数据集。考虑一下该从这份数据中提取什么样的特征。

检测数据是否包含服务中断问题的基本方法是看推文中是否使用了“延迟”一词。延迟事故涉及以下要素。

- 特定地点
- 特定时间
- 特定原因
- 特定时长

前 3 个要素可以在 Translink 的推文中持续追踪，但一些数据质量问题需要考虑。

“地点”体现在受影响的街道或站点的词语中（比如“在第 22 大街”）。就目的而言，这描述得不够完美，因为不做大量额外工作的情况下，不可能将街道名和路线起始点转化成一般的**受影响区域**（因为目前没有便捷方法可以基于这些信息绘制边界框）。

“时间”就是推文的发布时间，这也不够完美。虽然无法得知推文是否在发生服务中断后的固定时间内发布，但 Translink 的服务通知时间和发现问题的时间可能十分接近。这里不妨假设推文的发布时间是足够精确的，在这样的假设下继续研究。

当然也存在例外，比如某些长期存在的问题或者个别突发性问题（本以为微不足道，结果影响巨大的延迟事故）。在这种情况下，Translink 团队可能迟迟未发推文，直到他们意识到这场延迟需要发布到 Twitter 上。导致数据出现质量问题的另一个可能的原因是 Translink 内部沟通不畅，可能工程人员或平台团队不是每次都及时告知客服通知团队。

在没有真实、精确的 Translink 服务延迟数据集的情况下，只能粗略地度量这些延迟效应。（有的话当然更好。）

Translink 一般将“天空列车”服务延迟的原因分为以下几类。

- 铁路
- 列车
- 开关
- 控制
- 未知
- 干扰
- 医学
- 治安
- 电力

在 Twitter 正文中，每种原因都会用列表中的专用名词来描述。显然，一些原因（治安、电力、医药）和本示例关系不大，毕竟它们无法提供与路况相关的有用信息。火车行驶速度、路线和开关故障可能会导致绕行。这意味着，需要对这些实例进行分类。

同时，公交线路的服务延迟包含类似的编码，其中许多都很有用，具体如下所示。

- 机动车辆事故
- 建设
- 火灾
- 水管
- 交通

为这些事件类型编码会非常有用，尤其是有的服务延迟类型比其他的更具影响力，进而延长服务延迟的时间。需要为服务延迟类型编码，并将这些编码作为后续建模中的参数。

为此，需要对独热编码做一些改动，步骤如下所示。

- 为每种公交服务的风险类型构造一个条件变量，并初始化为 0。
- 检查每种公交服务的风险类型的推文内容。
- 将每个包含特定风险词的推文的相关条件变量值设为 1。

这样就无须执行烦琐的中间步骤（即创建需要处理的因子变量），高效地进行独热编码了。

```
from sklearn import preprocessing

enc = preprocessing.OneHotEncoder(categorical_features='all', dtype='float',
                                  handle_unknown='error', n_values='auto',
                                  sparse=True)

tweets.delayencode = enc.transform(tweets.delaytype).toarray()
```

除了每个事件的基础特征外，还可以观察服务中断问题和中断频率的关系。如果一周内出现了两次中断，那么第三次中断是更容易还是更不容易出现呢？

虽然这些问题很有趣，而且也可能对构建模型有所帮助，但通常更谨慎的做法是第一次建立一个有限的特征集和简单模型，而不是盲目地设计一个庞大的特征集。因此，用初始的发生率特征运行一次，并查看结果如何。

- **用户评论**

2010 年，文化领域的一个主流发展趋势是，有关个人观点表述的公共网络逐渐普及。这种现象产生了大量自述式的涉及多个领域的信息，可供我们使用。

通勤中断事故的频繁发生容易招致个人意见，从而在社交媒体上广泛传播。因此，如果编写好使用关键词检索的字典，就可以将 Twitter 作为全城交通问题的时间戳信息源。

为了收集这些数据，需要运用基于字典的查询方法。我们对这期间的大部分推文都不感兴趣（而且由于用的是 RESTful API，因此还需要考虑返回的数据限制问题），而关心如何识别出包含交通堵塞或延误相关关键词的推文数据。

然而，大量用户发布的推文往往格式不统一，不便于分析。因此，需要应用上一章介绍的技术，将数据分解成更易于分析的形式。

除了使用基于字典的查询，还要设法缩小查询范围。常规的方法是将由坐标构成的边界框作为 Twitter API 的参数，这样相关查询只会返回指定范围内的查询结果。

如前所示，第一次尝试会比较简单。本例将计算当前交通中断推文的数量。还可以对数据做一些额外处理，以便后续迭代。因为 Translink 数据包含定义清晰的延迟原因分类，所以可以试着利用专业字典（比如建设相关词汇及同义词字典）进行基于关键词的延迟类型分类。

还可以对中断事件在推文中的发生率定义更精细的量化标准，而不是像现在这样简单计数。例如可以构造一个加权计数特征，通过非线性加权提高同时发布的多条推文（可能代表严重中断事件）的影响。

- **必应交通 API**

下一个需要研究的 API 是必应交通 API。它易于使用、免费（一些同类竞争者的 API 需要付费），并且返回的内容足够细致。此外，该 API 还能返回事件地点代码、事件概况、堵塞信息、事件类型代码和起始时间戳。

该 API 提供的事件类型代码非常有用，能描述种类繁多的事件，具体如下所示。

(1) 事故。
(2) 堵塞。

(3) 残疾人车辆。
(4) 公共交通。
(5) 多方面。
(6) 其他新闻。
(7) 事先策划。
(8) 道路危险。
(9) 建设施工。
(10) 警报。
(11) 天气。

此外，该 API 还提供了严重程度代码，其对应的严重程度值如下所示。

(1) 影响极小。
(2) 影响较小。
(3) 影响中等。
(4) 影响巨大。

但该 API 有一个缺点：从各地区获取的信息不一致。例如在法国的查询会返回多个其他事件类型的代码（在法国北部的一个小镇，我一个月内观察到了 1、3、5、8 几个类型），但似乎不会出现所有代码种类。但在其他地区，即使数据量较少，也不会出现这种情况。温哥华常常只出现 9 或 5 类型的事件，但事件类型为“多方面”的事件实际多与建设相关。

```
Closed between Victoria Dr and Commercial Dr - Closed. Construction work. 5
```

该限制很棘手，而且不太容易处理，因为必应 API 并没有收集到我们想要的所有数据。除非付费购买更全面的数据集（或所在地区有数据更全的可用 API），不然只能勉强使用现有数据。

查询该 API 的示例如下所示。

```
importurllib.request, urllib.error, urllib.parse
import json

latN = str(49.310911)
latS = str(49.201444)
lonW = str(-123.225544)
lonE = str(-122.903931)

url = 'http://dev.virtualearth.net/REST/v1/Traffic/Incidents/'+latS+','+lonW+',
    '+latN+','+lonE+'?
key='GETYOUROWNKEYPLEASE'

response = urllib.request.urlopen(url).read()
data = json.loads(response.decode('utf8'))
resources = data['resourceSets'][0]['resources']

print('------------------------------------------------------')
```

```
print('PRETTIFIED RESULTS')
print('----------------------------------------------------')
for resourceItem in resources:
    description = resourceItem['description']
    typeof = resourceItem['type']
    start = resourceItem['start']
    end = resourceItem['end']
print('description:', description);
print('type:', typeof);
print('starttime:', start);
print('endtime:', end);
print('----------------------------------------------------')


This example yields the following data;

----------------------------------------------------
PRETTIFIED RESULTS
----------------------------------------------------
description: Closed between Boundary Rd and PierviewCres - Closed due to roadwork.
type: 9
severity 4
starttime: /Date(1458331200000)/
endtime: /Date(1466283600000)/
----------------------------------------------------
description: Closed between Commercial Dr and Victoria Dr - Closed due to roadwork.
type: 9
severity 4
starttime: /Date(1458327600000)/
endtime: /Date(1483218000000)/
----------------------------------------------------
description: Closed between Victoria Dr and Commercial Dr - Closed.
Construction work.
type: 5
severity 4
starttime: /Date(1461780543000)/
endtime: /Date(1481875140000)/
----------------------------------------------------
description: At Thurlow St - Roadwork.
type: 9
severity 3
starttime: /Date(1461780537000)/
endtime: /Date(1504112400000)/
----------------------------------------------------
```

即使 API 存在不同地区的事件类型代码不一致的情况，但其数据仍具有一些价值。交通中断事件的部分描述仍能提供一段时间内的数据。根据定义的区域交通事件的定位并返回对应时间的相关数据，很可能会提高模型的性能。

3. 用特征工程技术获取并选择变量

第一次处理输入数据时，仍保持初始特征集在较小规模。虽然数据中有许多信息值得分析，但暂不关注它们，而依旧将第一次的结果放在首位。

初始数据集可能无法高效地解决问题或者达到目标。此时，需要构建新特征并精简特征集（即仅保留特征构造结果中有价值的部分）来反复迭代特征集。

可以考虑使用独热编码和回归特征消除。本章将使用独热编码将天气数据和推文字典转化成大小为 $m*n$ 的张量。构造出 m 列新数据后，需要降低模型受到新特征误导的可能性（比如多个特征强调同一个信号，或者有误导性的常用词没有被第 6 章介绍过的数据清洗技术清洗掉）。前面讲过的特征选择技术**回归特征消除**非常适合执行此类任务。

总体而言，应用前两章介绍的方法（比如特征缩放）是非常有效的。首先，运用技术（比如转换和编码）生成具有潜在价值的新特征来扩大特征集，然后用技术识别出表现最佳的特征子集并删除表现不佳的那些特征。整个过程中测试不同的目标特征数量，以确定最佳可用特征集的特征数量。

不同的数据科学家会采取不同的做法。有的先反复迭代前面探讨过的特征创建技术来构造出所有特征，然后减小特征集，以最小化丢失数据的风险；有的则会反复执行整个过程。具体选择取决个人。

第一次处理输入数据时，特征集如下所示。

```
{
  'DisruptionInformation': {
    'Date': '15-05-2015',
    'TranslinkTwitter': [{
      'Service': '0',
      'DisruptionIncidentCount': '4'
  }, {
      'Service': '1',
      'DisruptionIncidentCount': '0'
    }]
  },
  'BingTrafficAPI': {
    'NewIncidentCount': '1',
    'SevereIncidentCount': '1',
    'IncidentCount': '3'
  },
  'ConsumerTwitter': {
    'DisruptionTweetCount': '4'
  }
}
```

该数据集看起来会表现不佳，但下面仍要基于它运行初始的基础算法，并初步了解与目标的距离。这样能以最低的成本快速了解数据。

方便起见，用一个非常简单的回归算法进行第一次迭代。技术越简单，运行速度就越快（而且往往更容易发现问题及其原因）。出于这个原因（还因为处理的是输出为连续型的回归问题，而不是分类问题），第一次迭代将建立一个简单的线性回归模型。

```
from sklearn import linear_model

tweets_X_train = tweets_X[:-20]
tweets_X_test = tweets_X[-20:]


tweets_y_train = tweets.target[:-20]
tweets_y_test = tweets.target[-20:]

regr = linear_model.LinearRegression()

regr.fit(tweets_X_train, tweets_y_train)

print('Coefficients: \n', regr.coef_)
print("Residual sum of squares: %.2f" % np.mean((regr.
predict(tweets_X_test) - tweets_y_test) ** 2))

print('Variance score: %.2f' % regr.score(tweets_X_test, tweets_y_test))

plt.scatter(tweets_X_test, tweets_y_test, color='black')
plt.plot(tweets_X_test, regr.predict(tweets_X_test),
         color='blue',linewidth=3)

plt.xticks(())
plt.yticks(())
plt.show()
```

此时的 AUC 还很差，只有 0.495，比目标低得多。下面输出一个混淆矩阵，看看模型究竟哪里表现不佳（见图 7-11）。

		实际结果	
		真	假
预测	真	1	9
	假	18	136

图　7-11

如上所示，各方面都不太理想。实际上，该矩阵意味着（模型认为）几乎所有记录都显示没有问题，也就是漏掉了 90%的实际中断事件。

实际上这还不算太糟，毕竟模型和特征都还处于初期，而且一些输入数据的价值也不太清楚。此外，事件发生率应有望达到 6%（因为训练数据表明，大约每 16 次通勤就会发生一次中断）。还应进一步假设每天都会出现一次通勤中断事件（如果忽略每天早早出门对生活方式的影响）。

考虑下一次迭代时的变化。

(1) 首先，进一步提升输入数据。运用一系列转换技术从现有数据源中提取一些新特征。
(2) 然后，用额外信息扩大数据集。其中描述温度和湿度的天气数据集也许有助于优化模型。

(3) 最后，可以升级算法，例如应用随机森林和支持向量机。但目前有充分的理由不这么做，因为可以从线性回归中继续学习不少信息。可以和先前的结果对比，了解改动后值增加了多少，同时保留快速的迭代循环和简单的评分方法。一旦做好特征准备并得到了最小有效特征集，就可以考虑升级模型了。

下面继续更新数据集。有多种可选方案。可以将地点编码进必应 API“描述”领域的交通事件数据和 Translink 推文中。对后者而言，这种做法对公交路线比对“天空列车”路线更有用（受限于分析范围，只能关注交通通勤）。

可以选用下面的方法来实现。

- 用街道名/地点的语料库解析输入数据，并构造独热矩阵。
- 简单地在全部 Twitter 正文和全部 API 数据上运行独热编码。

有趣的是，如果在独热编码后运用降维技术，就能在不出任何重大问题的情况下，对两种文本信息的全部文本进行编码。如果特征与 Twitter 文本中的其他词语不相关，就会被“回归特征消除”删除。

该方法灵活而巧妙。如果数据中含有其他潜在的有用内容，而一直以来被忽视未作为特征，这种处理就会基于这些信息构造新特征，从而增益。

下面用与延迟类型编码相同的方式给地区编码。

```
from sklearn import preprocessing

enc = preprocessing.OneHotEncoder(categorical_features='all', dtype='float',
                                  handle_unknown='error', n_values='auto',
                                  sparse=True)

tweets.delayencode = enc.transform(tweets.location).toarray()
```

另外，要继续根据 Translink 和必应地图的事件日志来构造近期的计数变量。此聚合过程的代码可从本章附带的 GitHub 库中下载。

用更新后的数据重新运行模型，所得结果有轻微提升：预测方差得分提升到了 0.56。虽然幅度不大，但表明模型在朝正确的方向迈进。

下面尝试第二种方法：添加新数据源获取天气数据。

- **天气 API**

前面得到的一些数据可表明通勤中断事件是否正在发生，即能实时反映现有延迟问题的数据源。下面做一些改变，寻找与延迟和堵塞原因相关的数据，例如施工信息和建设信息（这就引出了与必应交通 API 相关的一些代码）。

导致通勤时间延长的常见因素是糟糕的天气。有时这一点显而易见，比如浓雾和大风就对通勤时间有着显著影响。而在其他情况下，对特定通勤路线来说，天气因素与中断概率的关系的强弱和性质都不太明显。

通过提取粒度足够细、地理覆盖范围足够广的数据源中的相关天气数据，也许就能通过天气信息来改善中断事件的正确预测。

下面借助雅虎的天气 API 来实现。该 API 提供了当前和未来的温度、大气、压强以及其他天气数据。该 API 的查询无须密钥或登录过程，方法如下所示。

```
import urllib2, urllib, json

baseurl = https://query.yahooapis.com/v1/public/yql?

yql_query = "select item.condition from weather.forecast where
woeid=9807"
yql_url = baseurl + urllib.urlencode({'q':yql_query}) + "&format=json"
result = urllib2.urlopen(yql_url).read()
data = json.loads(result)
print data['query']['results']
```

为了了解 API 能提供什么，请将 `item.condition`（本质上是一个嵌入式 SQL 查询）替换成`*`。该查询输出了非常多的信息，而深入挖掘可以得到一些重要信息，包括当前天气状况。

```
{
  'channel': {
    'item': {
      'condition': {
        'date': 'Thu, 14 May 2015 03:00 AM PDT', 'text': 'Cloudy',
        'code': '26', 'temp': '46'
      }
    }
  }
}
```

7 日天气预报包含以下信息：

```
{
    'item': {
      'forecast': {
        'code': '39', 'text': 'Scattered Showers', 'high': '60',
        'low': '44', 'date': '16 May 2015', 'day': 'Sat'
      }
    }
}
```

以及当前其他天气信息：

```
'astronomy': {
     'sunset': '8:30 pm', 'sunrise': '5:36 am'

   'wind': {
     'direction': '270', 'speed': '4', 'chill': '46'
```

为了构建训练数据集,使用了一个自动化脚本来提取 2015 年 5 月到 2016 年 1 月的每日数据。其中的预测数据可能不那么有用，因为模型每天会在现有数据上重新运行，而不是依赖预测数据，但肯定会用到 wind.direction、wind.speed 和 wind.chill 变量，以及 condition.temperature 和 condition.text 变量。

至于如何进一步处理这些信息，有一种方法。对天气标签进行独热编码后，有关天气状况的信息就能转化成类型变量，就像上一章那样。这一步很必要。特征集得到了显著扩张，其中数据如下所示。

```
{
  'DisruptionInformation': {
    'Date': '15-05-2015',
    'TranslinkTwitter': [{
      'Service': '0',
      'DisruptionIncidentCount': '4'
    }, {
      'Service': '1',
      'DisruptionIncidentCount': '0'
    }]
  },
  'BingTrafficAPI': {
    'NewIncidentCount': '1',
    'SevereIncidentCount': '1',
    'IncidentCount': '3'
  },
  'ConsumerTwitter': {
    'DisruptionTweetCount': '4'
  },
  'YahooWeather':{
    'temp: '45'
    'tornado': '0',
    'tropical storm': '0',
    'hurricane': '0',
    'severe thunderstorms': '0',
    'thunderstorms': '0',
    'mixed rain and snow': '0',
    'mixed rain and sleet': '0',
    'mixed snow and sleet': '0',
'freezing drizzle': '0',
'drizzle': '0',
'freezing rain': '0',
'showers': '0',
'snow flurries': '0',
'light snow showers': '0',
'blowing snow': '0',
'snow': '0',
'hail': '0',
'sleet': '0',
'dust': '0',
'foggy': '0',
'haze': '0',
```

```
'smoky': '0',
'blustery': '0',
'windy': '0',
'cold': '0',
'cloudy': '1',
'mostly cloudy (night)': '0',
'mostly cloudy (day)': '0',
'partly cloudy (night)': '0',
'partly cloudy (day)': '0',
'clear (night)': '0',
'sunny': '0',
'fair (night)': '0',
'fair (day)': '0',
'mixed rain and hail': '0',
'hot': '0',
'isolated thunderstorms': '0',
'scattered thunderstorms': '0',
'scattered showers': '0',
'heavy snow': '0',
'scattered snow showers': '0',
'partly cloudy': '0',
'thundershowers': '0',
'snow showers': '0',
'isolated thundershowers': '0',
'not available': '0',
}
```

可能要花大量时间进一步丰富雅虎天气 API 提供的数据。如前所示，第一次运行依旧采用之前所讲的模型并输入特征。

如何进一步处理这些数据值得思考。本例中，重要的是区分跨列数据转换和跨行数据转换。

跨列转换指的是，相同输入案例中的不同特征变量基于另一个变量进行转换。例如可以取实例的开始日期和结束日期，计算持续时长。而本书介绍的大多数算法技术并没有从这样的转换中获益。能构建非线性决策边界的大多数机器学习技术往往会在对数据集建模的过程中实现变量间的关系编码。深度学习技术则会进一步发挥这种能力。这就是一些特征工程技术（尤其是基础的数据转换）不会为深度学习应用功能带来多少提升的部分原因。

跨行转换的一个典型代表是数据聚合。例如最后的 n 个持续时间值的集中趋势就是一个特征，通过对多行进行计算得出。显然，一些特征可以通过一系列逐列计算和逐行计算的组合得到。跨行转换的特别之处是，模型往往很难通过训练识别这种特征，即跨行转换产生的特征更可能在特定情况下带来价值。

当然，这些信息之所以有意义，是因为对近期天气进行跨行运算后得出的特征也许能给模型带来新信息。例如最近 n 小时内的气压或温度变化可能比当前气压或温度更有用。（尤其是模型要预测当天晚些时候的通勤情况时。）

然后重新运行模型。这一次的 AUC 略有提高，到了 0.534。混淆矩阵同样有所改善（见图 7-12）。

		实际结果	
		真	假
预测	真	3	7
	假	22	132

图 7-12

如果问题和天气因素有关，那么可以继续研究天气数据。增大方案考虑的时间跨度，将从各个数据源积累更多纵向数据，从而做出更可靠的预测。

此时，最终目标已经近在咫尺了！可以继续扩大输入数据集，但诉诸其他方法更为明智。这里有两种比较有效的方案。

数据科学家往往会考虑简化假设。惯常的做法是将帕累托原理（Pareto principle）应用于成本效益分析决策。帕累托原理从根本上阐述了大多数情况遵循帕累托分布，即约 80%的价值（或影响）来自约 20%的输入（或原因）。这个概念在软件工程中最为常用，因为可以作为指导来提高效率（见图 7-13）。

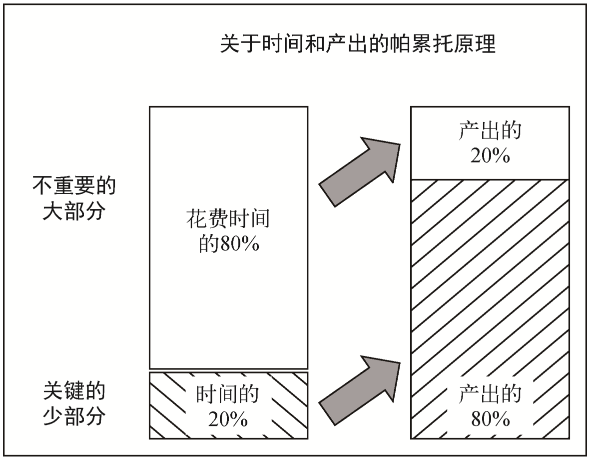


图 7-13

为了将理论付诸实践，应该花更多时间细化特征工程，因为还有一些技术没有用过，还有一些特征可以构造。此外，还有一些方法仍这里未涉及，尤其是外部数据搜索和模型切换，可以简单尝试一下。下一次运行时，可以先尝试这些试错成本低但可能意义重大的方法，再钻研额外的数据准备，这样做比较明智。

在进行探索性分析的过程中，有的变量非常稀疏。目前还不清楚它们是否有用（尤其是对站点来说，某些特定事件很少发生）。

下面用本章介绍过的技术测试一下变量集。对问题应用 Lasso，将特征集简化为表现更佳的子集。

```
fromsklearn.preprocessing import StandardScaler


scaler = StandardScaler()
X = scaler.fit_transform(DisruptionInformation["data"])
Y = DisruptionInformation["target"]
names = DisruptionInformation["feature_names"]

lasso = Lasso(alpha=.3)
lasso.fit(X, Y)

print "Lasso model: ", pretty_print_linear(lasso.coef_, names, sort = True)
```

这个输出的价值立即可见。显然，许多天气特征（有的没有充分表现出来，有的没有给出对模型有用的信息）对模型并无作用，需要删除。此外，对交通的聚合并未产生多少价值。虽然可以暂时保留它们（但愿数据量的提升能增强它们的作用），但下次运行模型时将删掉 LASSO 暴露出的评分较低的那些特征。

这里可以做一个试验成本很低的尝试：升级模型，使其能够进行非线性拟合，从而逼近任何函数。这个改变很值得尝试，因为一些特征表现出了一系列偏斜分布，这表示存在潜在的非线性趋势。下面对该数据集应用随机森林，如下所示。

```
fromsklearn.ensemble import RandomForestClassifier,
ExtraTreesClassifier
rf = RandomForestRegressor(n_jobs = 3, verbose = 3, n_estimators=20)
rf.fit(DisruptionInformation_train.targets,DisruptionInformation_train.data)

r2 = r2_score(DisruptionInformation.data, rf.predict(
DisruptionInformation.targets))
mse = np.mean((DisruptionInformation.data - rf.predict(
DisruptionInformation.targets))**2)

pl.scatter(DisruptionInformation.data, rf.predict(
DisruptionInformation.targets))
pl.plot(np.arange(8, 15), np.arange(8, 15), label="r^2=" + str(r2),
        c="r")
pl.legend(loc="lower right")
pl.title("RandomForest Regression with scikit-learn")
pl.show()
```

然后再看看混淆矩阵（见图 7-14）

		实际结果	
		真	假
预测	真	4	6
	假	15	134

图 7-14

效果相当不错，模型的简单升级带来了显著提升。虽然对于少量实例，模型未能正确分类，但已能正确识别出近 40%的通勤延迟事件了（变得有用了）。

然而该模型还是会给出错误的指导（会导致不必要的早起）。当然，判断模型质量的标准是，模型正确预测的通勤延误是否比导致的错误（不必要的早起）更多。如果继续在一段时间内收集特征数据，模型就可能达到该目标。该模型的主要缺点是可取样的实例太少，毕竟通勤中断事件本身就很少。

但已经成功从不同数据源收集并整理了一系列数据，通过免费可用的数据构建模型，带来了可见的现实收益（使上班迟到的次数降低了 40%）。这当然是值得高兴的成果！

7.4 扩展阅读

关于特征选择入门，推荐阅读 Ando Sabaas 对各种特征选择技术的四步探索（http://blog.datadive.net/selecting-good-features-part-i-univariate-selection/）}。文中包含 Python 代码片段和清晰易懂的注释。

有关第 6 章和第 7 章对特征选择和特征工程的讨论，参见 Alexandre Bourhard-Côté 的文档（http://people.eecs.berkeley.edu/~jordan/courses/294-fall09/lectures/feature/slides.pdf）以及 Jeff Howbert 的文档（http://courses.washington.edu/css490/2012.Winter/lecture_slides/05a_feature_creation_selection.pdf）。

本章没有涵盖特征构造的方方面面。许多资料讨论的多是降维技术和特定领域的特征构造技术。建议阅读代码文档，以全面了解各种特征转换技术。在现有知识的基础上，可以阅读 Spark ML 的特征转换算法文档（https://spark.apache.org/docs/1.5.1/ml-features.html#feature-transformers），它描述了针对数值型特征和文本特征的多种特征转换技术。但要记住，特征构造往往针对特定问题和特定领域，而且创造性极强。当掌握了一系列技术后，接下来要做的就是针对当前问题合理应用这些技术。

如果对超参数优化感兴趣，建议阅读 Alice Zheng 在 Turi 的博客上发表的文章（How to Evaluate Machine Learning Models, Part 4: Hyperparameter Tuning），这是一个不错的开端。

关于网格搜索技术，可以参考 scikit-learn 的文档。

7.5 小结

本章介绍并应用了一些技术。这些强大的技术可以基于极少的初始数据，为机器学习高效构建数据集并进行处理，还能使看似简单的数据集迸发出无限可能。本章用一些客服相关推文构造了一个通勤中断预测器，展示了这些技术的强大。

为了使解决方案切实可用，还需要添加一些功能。删除倒数第二步中的一些地址的做法有点问题。如果解决方案是为了预知交通中断风险，那删除地点的做法似乎并不可取，特别是缺乏全年数据，因而无法识别季节性或纵向趋势（比如长期的维护工作或计划内的站点关闭）时更是如此。删除这些元素有些草率，建议多保留一段时间。

出于这些顾虑，我们应该意识到，在解决方案中引入动态机制很有必要。当春天再次到来，数据集开始包含新的天气条件时，模型很可能无法像以前那样高效适应了。下一章将介绍如何构建更复杂的集成模型，以及提升模型稳健性的方法。

第 8 章

集成方法

8

前几章介绍了一些新技术，包括几种高级机器学习算法和许多配套技术，可实现高效的特征选择和特征准备。本章旨在用集成方法增强现有的技术。集成方法是指将多个模型组合在一起，以解决实际问题。

集成技术现已是数据科学家的常用方法。在机器学习技术百花竞放的背景下，使用集成方法已成常态，而且集成方法对于很多领域都不可或缺。本章将介绍的技术可以大幅提升模型性能，并提升模型面对潜在数据变化的稳健性。

本章将介绍一系列集成技术，探讨这些技术的代码和应用，还将以现实应用作为参考来深化内容讲解，其中包括 Kaggle 优胜者构建的模型。

本章开发的各个模型都有助于解决许多数据问题，但将模型应用于生产会产生一些额外的问题，所得的解还是容易受到基础观测改变的影响。如果在不同群体、时间变化（比如有季节性变化的现象）或基础条件变化中，模型结果表现一致，这说明模型在有限的数据条件下实现了很好的泛化，并能够保持优越的性能。

本章最后会介绍如何将本书所讲的技术付诸实践，以及如果希望应用能够适应变化，那么需要考虑保持额外的监测和维护。

8.1 集成简介

> “赢得机器学习比赛的方法是：将他人的成果集成其中。”
>
> ——Vitaly Kuznetsov NIPS2014

在机器学习中，集成模型指的是用于解决同一个问题的一套模型。它包含两部分：一组模型以及设定的如何将多个模型结果进行融合的一套规则。

针对给定问题，数据科学家设计出多种解决方案，取每种方案之精髓，构造出最终结果，这就是集成。这样做提高了模型应对噪声的稳健性，这一点表现在模型在初始数据集上更有效地进

行训练（降低过度拟合的可能性并减少训练误差），也更能应对前面提及的数据变化问题。

毫不夸张地说，集成算法是机器学习领域近期最重要的进展。

另外，集成算法有助于灵活解决问题，因为它允许数据科学家测试解的不同部分，并解决针对输入数据子集或所使用模型的一部分的问题，而无须调整整个模型。稍后将看到，这大大简化了工作。

根据使用的决策规则性质，集成算法通常可分为以下几类。

- **平均法**：并行构建多个模型，然后用平均或投票技术生成集成估计器。
- **堆叠法（或混合法）**：将多个分类器的输出进行加权，作为下一层模型的输入项。
- **提升法**：依次构建模型，添加的每个模型都以提高集成估计器的评分为目标。

根据这几种集成方法的重要性和实用性，下面依次说明：原理讲解、算法选择和实际案例。

8.1.1　理解平均集成

平均集成在自然科学和统计建模领域应用已久，常见应用包括分子动力学和声频信号处理。这种集成法通常能很好地反映给定系统的完整情况。各情况间的均值和方差是整个系统的关键值。

在机器学习领域，平均集成指的是在同一个数据集上训练的一系列模型，结果以多种方式聚合。根据实现目标的不同，平均集成可以带来许多好处。

平均集成可以缓解模型表现的不稳定。常用方法包括设计多个模型配置方案，每一个都将不同的参数子集作为输入项。采用该方法的技术统称为“有放回取样算法”（bagging）。

1. 使用 bagging

不同 bagging 实现的运行方法不同，但都使用特征空间的随机子集。bagging 方法主要有 4 类。无放回取样（pasting）是指提取样本的随机子集而不再放回。如果放回，就叫作 bagging。

将实例按照特征进行采样时，该方法叫作**随机子集**。随机子集法的功能稍有不同，可以让我们免于高度优化大量特征选择。这种转变使得单模型能拥有最佳输入集，而随机子集法则允许多种模型配置方案并行，然后将每个解的方差进行平均。

> 虽然集成缓解模型性能不稳定的做法似乎会影响模型性能，（很多人对此的直接反应是，为什么不只是找出集成中表现最佳的那个模型？）但这种方法其实还有很多优点。
>
> 首先，如前所述，取平均能提升模型集适应陌生噪声的能力（即降低过度拟合）。其次，集成模型可以针对输入数据集的不同特点高效建模。这是机器学习大赛中的常用方法，数据科学家会基于分类结果和失败案例的特定类型来反复调整集成。有时检验模型结果需要考虑周全（通常作为正常迭代模型开发过程的一部分），但许多数据科学家会有一些偏爱的、会优先采用的技巧或解决方案。

随机子集法非常强大，尤其是可以用大量子集来详尽地检验特征的组合。但随机子集法的使用成本随数据集规模的增大而呈非线性增长，超过某个点后，对于构造大量子集的算法来说，测试每种模型配置方案的成本将非常高昂。

最后，可以使用一种名为**随机补丁**的方法，该方法根据从样本和特征中提取的子集构成集成估计器。在类似的情况下，随机补丁的效果往往和随机子集技术的效果相近，但能大幅减少内存消耗。

介绍完 bagging 集成方法背后的原理后，下面着手实现。以下代码描述了用 sklearn 的 `BaggingClassifier` 类实现的基于随机补丁的分类器。

```
from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)
X = data
y = digits.target


bagging = BaggingClassifier(KNeighborsClassifier(), max_samples=0.5,
                            max_features=0.5)
scores = cross_val_score(bagging, X, y)
mean = scores.mean()
print(scores)
print(mean)
```


与许多 sklearn 分类器相同，所需要的核心代码非常简单，能够初始化分类器并对数据集评分即可。交叉验证（通过 `cross_val_score` 实现）对计算复杂度的影响很小。

bagging 分类器以 *K* 近邻分类器（`KNeighboursClassifier`）为基础，并将针对特征和案例的抽样率均设置为 50%。在手写数字数据集上取得了优异的成绩：交叉验证后，数据的平均分类正确率达到了 93%。

```
[ 0.94019934 0.92320534 0.9295302 ]

0.930978293043
```

2. 使用随机森林

平均集成技术的替代方法统称为“随机森林”。它可能是参加比赛的数据科学家使用的最成功的集成技术。该技术可以实现并行的几套决策树分类器。对分类器构造引入两种主要的随机性[①]后，森林最终会包含不同的树。用于构造每棵树的数据是对训练集进行有放回抽样得到的，同时，树的构造过程不再使用所有特征的最佳分割，而是选择特征随机子集的最佳分割。

可以用 sklearn 的 `RandomForestClassifier` 类轻松调用随机森林。如以下代码所示。

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)

n_samples, n_features = data.shape
n_digits = len(np.unique(digits.target))
labels = digits.target

clf = RandomForestClassifier(n_estimators=10)
clf = clf.fit(data, labels)
scores = clf.score(data,labels)
print(scores)
```

该集成模型输出的分数为 0.999，这很惊人。确实，前几章应用的任何单个模型从未达到过这样的效果。

随机森林的变体**极端随机树**使用相同的特征随机子集法，在树的每个分支处选择最佳分割，然而该方法还会将分类的阈值随机化。决策树往往会选择类间最有效的分割，极端随机树则会按随机值进行分割。

由于决策树的训练相对高效，随机森林算法的内部会生成大量树，这使得分类器可以随着节点数的增加而提高性能。引入的随机性能增强模型对噪声或数据变化的稳健性。但像前面讲过的 bagging 算法那样，这种稳健性的提升往往伴随着效果的轻微下降。对极端随机树而言，随着稳健性的增强，模型的性能测度会进一步提高（通常是偏差值降低）。

以下代码展示了极端随机树在实践中的应用方法。这段代码与随机子集实现一样简单。下面建立一组模型来比较极端随机树和随机森林在树的生成方面的区别。

① 即实例的随机性和特征的随机性。——译者注

```
from sklearn.cross_validation import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_digits
from sklearn.preprocessing import scale

digits = load_digits()
data = scale(digits.data)
X = data
y = digits.target

clf = DecisionTreeClassifier(max_depth=None, min_samples_split=1,
                             random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)

clf = RandomForestClassifier(n_estimators=10, max_depth=None,
                             min_samples_split=1, random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)

clf = ExtraTreesClassifier(n_estimators=10, max_depth=None,
                           min_samples_split=1, random_state=0)
scores = cross_val_score(clf, X, y)
print(scores)
```

分数如下所示。

```
[ 0.74252492  0.82136895  0.75671141]
[ 0.88372093  0.9015025   0.8909396 ]
[ 0.91694352  0.93489149  0.91778523]
```

由于应用的是完全基于树的方法，评分仅表示标记正确的实例比例。可以看到，两种森林方法区别不大，表现都极佳，平均分数为 0.9。在本例中，随机森林其实险胜极端随机树（约高出 0.002），但两种技术都明显优于基础决策树（平均分数为 0.77）。

随机森林的一个缺点（尤其随着森林规模的增大）是，检验某次模型的实现或微调的效果会很难。虽然一棵树容易处理，但集成算法所包含的数量众多的决策树和随机分割引起的困惑，会令随机森林的实现难以优化。克服该缺点的方法之一是，关注单个模型绘制的决策边界。通过比较集成算法包含的所有模型，便于判断一个模型能否比其他模型分类更准确。

对于本例，无须深究具体细节，就能通过一种巧妙的方式轻松了解模型的效果（见图 8-1）。

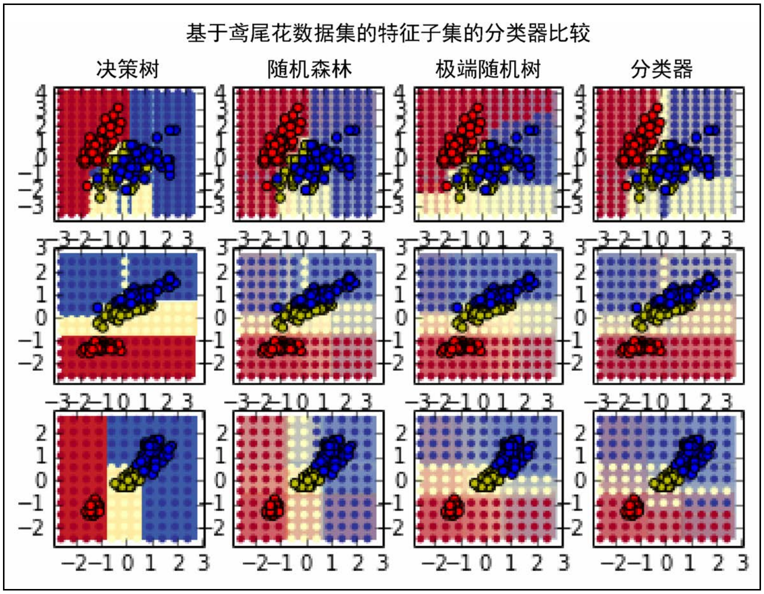


图　8-1

尽管在更高层次（即通过高级绘图技巧和汇总得分情况）理解随机森林实现的效果不太容易，但付出是值得的。随机森林的效果非常好，只需要在计算方面多投入一点成本。该技术适用于问题早期，有助于确定方向和手段，因为该技术能立即产生强有力的结果，可以提供一个有用的基准。介绍完如何实现随机森林后，下面可以优化并扩展集成算法了。

应继续探索不同的集成技术，丰富自己的集成方法。

8.1.2　应用提升法

构建集成模型的另一种方法是构建 boosting 模型，即依次使用多个模型，迭代地“提升”或改善集成模型的效果。

boosting 模型常用一些弱学习器（即仅比随机猜测效果稍好一点的模型）。每次迭代时，一个新的弱学习器会在调整后的数据集上进行训练。经过多次迭代后，集成算法会在每次迭代时生成一棵新树（可用于提高集成模型的性能得分）。

AdaBoost 可能是最知名的 boosting 法了。通过以下方法，它能在每次迭代时调整数据集。

- 选择一个决策树桩（即通常只有一层的浅决策树，本质上是问题中的数据集的最显著决策边界）。
- 增大决策树桩错误标签的实例的权重，同时降低正确标签的实例的权重。

该迭代的权重调整过程导致集成模型中的每个新分类器都会优先训练（上次迭代）标签错误的实例。模型根据针对高权重的数据点进行调整。最后，所有决策树桩合并成最终的分类器。

AdaBoost 既能用于分类，又能用于回归，而且效果都十分惊艳。以下示例展示了 AdaBoost 在 heart 数据集上的应用。

```
import numpy as np

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets.mldata import fetch_mldata
from sklearn.cross_validation import cross_val_score

n_estimators = 400
# A learning rate of 1. may not be optimal for both SAMME and SAMME.R
learning_rate = 1.

heart = fetch_mldata("heart")
X = heart.data
y = np.copy(heart.target)
y[y==-1]=0

X_test, y_test = X[189:], y[189:]
X_train, y_train = X[:189], y[:189]

dt_stump = DecisionTreeClassifier(max_depth=1, min_samples_leaf=1)
dt_stump.fit(X_train, y_train)
dt_stump_err = 1.0 - dt_stump.score(X_test, y_test)

dt = DecisionTreeClassifier(max_depth=9, min_samples_leaf=1)
dt.fit(X_train, y_train)
dt_err = 1.0 - dt.score(X_test, y_test)

ada_discrete = AdaBoostClassifier(
    base_estimator=dt_stump,
    learning_rate=learning_rate,
    n_estimators=n_estimators,
    algorithm="SAMME")
ada_discrete.fit(X_train, y_train)

scores = cross_val_score(ada_discrete, X_test, y_test)
print(scores)
means = scores.mean()
print(means)
```

在本例中，n_estimators 参数决定所需的弱学习器的数量。如果使用的是平均法，那么添加估计器会一直降低模型的偏差，同时会提升模型对训练数据过度拟合的概率。base_estimator 参数可以定义不同的弱学习器，默认为决策树（由于训练弱树非常简单，可以使用树桩，即非常浅的树）。如本例所示，将 AdaBoost 应用于 heart 数据集时，AdaBoost 的正确标签率达到了 79%，对于第一次尝试来说相当不错了。

```
[ 0.77777778  0.81481481  0.77777778]

0.79012345679
```

boosting 模型有着平均模型不具备的显著优势：它简化了识别问题实例及其类型并进行解决的过程。boosting 模型往往先处理最容易预测的实例，每次加入的新模型则拟合剩下的预测错误的实例所构成的子集。

这伴随着一个风险：boosting 法开始过度拟合（举个极端的例子，集成模型的每个组件已经完全拟合了特定实例）训练数据了。控制集成模型组件在正确数量是一个棘手的问题，可以通过一项熟悉的技术来解决。第 1 章介绍过一种名为**“肘部法”**的视觉启发式方法。 当时的例子用图像表示 *K*（均值个数）和聚类效果的关系。这里可以用估计器的数量（*n*）和集成模型的偏差率或误差率（下文称为 *e*）来执行一个类似的过程。针对不同的 boosting 估计器，可以绘制出它们的输出，如图 8-2 所示。

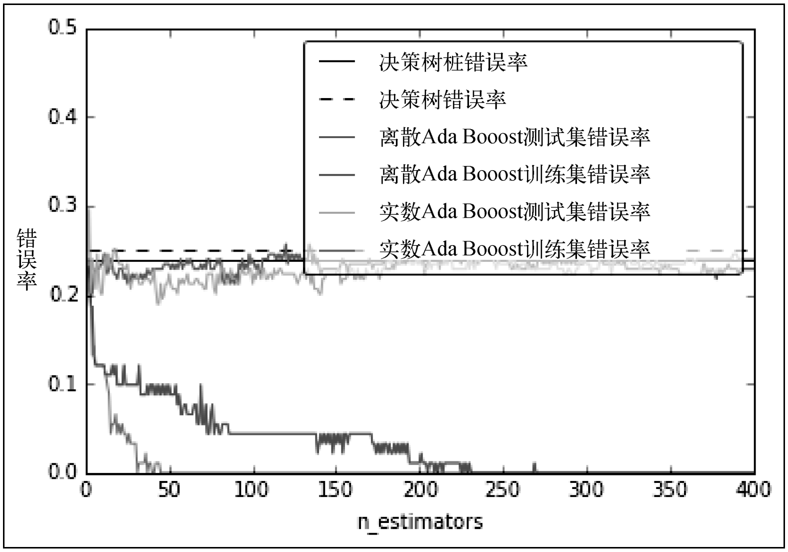


图　8-2

确定曲线开始趋平的点，有助于降低模型过度拟合的风险。因为曲线开始趋平时，过度拟合风险出现的可能性也会增长。原因其实很简单：趋平的曲线代表每个新增的估计器产生的收益越来越少，即成功分类的实例越来越少了。

这种视觉辅助有助于展现解出现过度拟合的可能性。可以（而且应该）尽可能地应用验证技术，但在某些情况下（比如实现了模型并想评估其是否达到特定的 MVP 目标时，可以通过实例或 Kaggle 公共排行榜上的分数分布获知），有人可能会忍不住在一个表现不错的模型上继续。添加新的估计器时，要准确理解所获收益是如何减少的，这对于理解过度拟合的风险至关重要。

使用 XGboost

2015 年年中，出现了一种解决结构化机器学习问题的新算法：XGboost（extreme gradient boosting，极端梯度提升法），该算法迅速征服了数据科学竞赛领域。它是一个写得好、表现佳的库，提供了一种泛化的 boosting 算法（梯度提升法）。

XGBoost 的运行很像 AdaBoost，但两者有一个关键区别：改进模型的方法不同。

每次迭代时，通过减少集成模型的残差（目标和标签预测的差），XGboost 力图提升现有模型集合的效果，以模型能否最大程度降低现有集成模型的残差为基础来进行添加。这与梯度下降法类似（在梯度下降法中，函数沿梯度下降的反方向移动而迭代地最小化），因此得名“梯度提升”。

事实证明，梯度提升法在近几年的 Kaggle 大赛中表现出众。CrowdFlower 大赛和微软恶意软件分类大赛，以及 2015 年下半年许多其他结构化数据比赛的优胜者都采用了该方法。

为了应用 XGboost，需要先获取 XGboost 库。最佳方法是通过 `pip`，在命令行输入 `pip install xgboost`。对 Windows 用户来说，Windows 目前已无法安装 `pip`。方便起见，可在本书的 GitHub 库的 Chapter 8 文件夹中获取 XGboost 的冷备份。

XGboost 易于使用，本例将该库应用于一个多类别的分类任务中：UCI 皮肤病学数据集（UCI dermatology dataset）。该数据集包含年龄变量和大量类型变量。取一行数据为例，如下所示。

```
3,2,0,2,0,0,0,0,0,0,0,0,1,2,0,2,1,1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,10,2
```

年龄值（倒数第二个特征）有少量缺失，编码为`?`。处理该数据集旨在判断皮肤状况属于 6 类中的哪一种，6 类皮肤状况的分布如下所示。

```
Database:  Dermatology

Class code:   Class:                  Number of instances:
1             psoriasis      112
2             seboreic dermatitis             61
3             lichen planus                   72
4             pityriasis rosea                49
5             cronic dermatitis               52
6             pityriasis rubra pilaris        20
```

加载数据，并按 7∶3 的比例将数据分割成训练集和测试集，对该问题应用 XGboost。

```
import numpy as np
import xgboost as xgb

data = np.loadtxt('./dermatology.data', delimiter=',',
                  converters={33: lambda x:int(x == '?'), 34: lambda x:int(x)-1 } )
sz = data.shape
```

8

```
train = data[:int(sz[0] * 0.7), :]
test = data[int(sz[0] * 0.7):, :]


train_X = train[:,0:33]
train_Y = train[:, 34]


test_X = test[:,0:33]
test_Y = test[:, 34]
```

这样就对模型进行了初始化和参数配置。eta参数定义了步长收缩。在梯度下降算法中，收缩参数常用于确定一次更新的规模[①]。梯度下降算法容易（尤其是接近收敛时）在最优值附近上下波动，通过收缩参数来降低变动规模，可以使梯度下降结果更精准。常用的（默认）值是 0.3，本例中的eta设置为了 0.1，以便精度更高（但可能需要更多次迭代）。

max_depth 参数直观地定义了本例中所有树的最大深度。鉴于模型有 6 类输出，所以 6 是个不错的初始值。num_round 参数定义了算法将执行梯度提升法的次数。如前所述，多类问题的类别数越多，需要执行梯度提升法的次数往往也就越多。nthread 参数则定义了代码将在多少个 CPU 线程上运行。

这里使用 DMatrix 结构意在提升训练速度并优化内存。使用 XGboost 时也可以考虑使用该结构，可用 numpy.arrays 构建。DMatrix 能激活 watchlist 的功能，解锁一些高级特性。借助 watchlist 能监控所提供列表中所有数据的评估结果。

```
xg_train = xgb.DMatrix( train_X, label=train_Y)
xg_test = xgb.DMatrix(test_X, label=test_Y)

param = {}

param['objective'] = 'multi:softmax'

param['eta'] = 0.1
param['max_depth'] = 6
param['nthread'] = 4
param['num_class'] = 6

watchlist = [ (xg_train,'train'), (xg_test, 'test') ]
num_round = 5
bst = xgb.train(param, xg_train, num_round, watchlist );
```

训练模型 bst 来生成一个初始预测。然后重复训练过程，并用 softmax（通过 multi:softprob）生成一个预测。

```
pred = bst.predict( xg_test );
```

① 与学习速率相似。——译者注

```
print ('predicting, classification error=%f' % (sum( int(pred[i]) != test_Y[i]
       for i in range(len(test_Y))) / float(len(test_Y)) ))


param['objective'] = 'multi:softprob'
bst = xgb.train(param, xg_train, num_round, watchlist );

yprob = bst.predict( xg_test ).reshape( test_Y.shape[0], 6 )
ylabel = np.argmax(yprob, axis=1)

print ('predicting, classification error=%f' % (sum( int(ylabel[i]) != test_Y[i]
       for i in range(len(test_Y))) / float(len(test_Y)) ))
```

8.1.3　使用堆叠集成

前面的传统集成模型遵循了共同的设计原则：包含多个分类器，这些分类器为拟合一系列目标标签而进行训练，以及通过模型投票和模型提升等策略生成一些元函数的模型。

集成构建还有一种设计原则，称作“堆叠”或“混合”。堆叠涉及多个模型组成的层结构，其中一层模型的输出用作下一层模型的训练数据。混合上百个不同模型也是可能实现的。

堆叠集成也能混合多个子混合输出构成的特征集（也称**多重混合**）。还可以在不同级别的混合或子混合中，从堆叠集成的模型中提取高效的参数，将其用作元特征。

所有这些造就了堆叠集成技术强大且可扩充。Kaggle Netflix 大奖（比赛奖金为 100 万美元）的获得者对上百个特征运用堆叠集成，成效显著。他们还借助了其他技巧来提升预测效果。

- 他们训练并优化了集成模型，同时保留了一些数据。对测试集应用模型前，先用保留数据对集成模型重新进行训练和优化。这种做法并不鲜见，但效果很好，因此值得尝试。
- 他们利用梯度下降法进行训练并将均方根误差作为性能评测函数。关键是，他们将集成的均方根误差（而不是其中任何模型的均方根误差）用作相关性能指标（残差测度）。无论何时使用集成，都可以考虑这样做。
- 他们使用了模型组合来减少其他模型残差，比如基于近邻的方法就可以减少受限玻尔兹曼机的残差，前面介绍过。通过摸索机器学习算法的相对优势和劣势，可以找到理想的集成配置。
- 他们用 k 重交叉验证计算了混合的残差，前几章介绍并应用过该技术。这避免了仅使用同一数据集训练模型并进行混合的问题。

赢得 Netflix 大奖的 Pragmatic Chaos 模型主要胜在可高度定制。该模型的出色表现源于密集迭代和创造性的网络配置优化。另一个关键点是堆叠集成的基本架构模式，如图 8-3 所示。

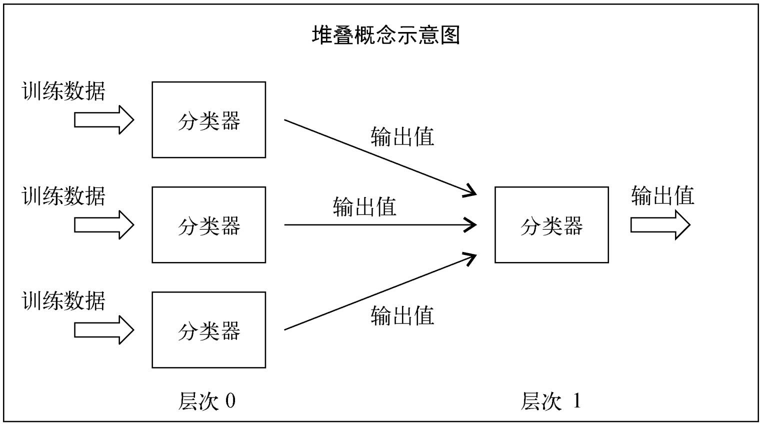


图　8-3

介绍过堆叠集成运行的基础知识后，下面试着用它解决数据问题。首先要使用 GitHub 库中 Chapter 8 文件夹下的 `blend.py` 代码。Kaggle 大赛的高分获得者就使用了该混合代码的不同版本。

首先看看堆叠集成如何解决现实的数据科学问题：Kaggle 大赛**预测生物反应**的目标是基于分子的化学性质构建模型，以预测其生物反应，该模型应尽可能高效。下面透过大赛中的杰出参赛者来了解堆叠集成在实际应用中的效果。

在数据集中，每一行代表一种分子，每种分子又有 1776 种描述其特性的特征。这里的目标是基于分子的性质预测二元反应。

下面即将应用的代码来自比赛中的一位参赛者，他用堆叠集成合并了 5 种分类器：2 种配置不同的随机森林分类器、2 种极端随机树分类器，以及 1 种梯度提升分类器。其中梯度提升分类器的预测与其他 4 种分类器略有区别。

重复的分类器的分割原则稍有不同。它使用了**基尼不纯度**（Gini Impurity，gini），基于问题的潜在标签分布对记录进行随机标记，然后度量一条随机记录被错误标记的频率。另一棵树使用了信息增益（熵），这是一种信息量测度。潜在分支的信息量可以通过编码它的比特数来度量。用熵来判断合适的分支会降低分支的多样性，但值得注意的是，熵和 gini 标准产生的结果可能大不相同。

```
if __name__ == '__main__':

    np.random.seed(0)

    n_folds = 10
    verbose = True
    shuffle = False
```

```
X, y, X_submission = load_data.load()

if shuffle:
    idx = np.random.permutation(y.size)
    X = X[idx]
    y = y[idx]

skf = list(StratifiedKFold(y, n_folds))

clfs = [RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='gini'),
        RandomForestClassifier(n_estimators=100, n_jobs=-1,
                               criterion='entropy'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                             criterion='gini'),
        ExtraTreesClassifier(n_estimators=100, n_jobs=-1,
                             criterion='entropy'),
        GradientBoostingClassifier(learning_rate=0.05, subsample=0.5,
                                   max_depth=6, n_estimators=50)]

print "Creating train and test sets for blending."

dataset_blend_train = np.zeros((X.shape[0], len(clfs)))
dataset_blend_test = np.zeros((X_submission.shape[0], len(clfs)))

for j, clf in enumerate(clfs):
    print j, clf
    dataset_blend_test_j = np.zeros((X_submission.shape[0], len(skf)))
    for i, (train, test) in enumerate(skf):
        print "Fold", i
        X_train = X[train]
        y_train = y[train]
        X_test = X[test]
        y_test = y[test]
        clf.fit(X_train, y_train)
        y_submission = clf.predict_proba(X_test)[:,1]
        dataset_blend_train[test, j] = y_submission
        dataset_blend_test_j[:, i] = clf.predict_proba(X_submission)[:,1]
    dataset_blend_test[:,j] = dataset_blend_test_j.mean(1)

print
print "Blending."
clf = LogisticRegression()
clf.fit(dataset_blend_train, y)
y_submission = clf.predict_proba(dataset_blend_test)[:,1]

print "Linear stretch of predictions to [0,1]"
y_submission = (y_submission - y_submission.min()) /
                (y_submission.max() - y_submission.min())

print "Saving Results."
np.savetxt(fname='test.csv', X=y_submission, fmt='%0.9f')
```

如果将结果提交到个人排行榜，会发现名次相当不错：第 12 名（共有 699 名参赛者）！当然，比赛早已结束，事后的成绩不够客观，但鉴于代码很简洁，这个结果仍非常惊艳！

在实践中应用集成

应用集成时需要铭记在心的是，目标是调整集成的效果，而不是调整构成集成的模型的效果。因此，所用方法应注重构建效果良好的集成模型，而不是构造最强的单个模型。

对集成中的模型的关注度因人而异。建议尝试配置不同或初始化不同的同类模型（比如随机森林），并重点关注集成和构成集成的元参数所产生的效果。

对于更棘手的问题，往往需要密切关注集成中的单个模型。试着构建更小的集成以解决更棘手的问题时，这一点显然是正确的，但要想构建优良的集成，往往需要考虑结构背后的参数和算法。

如前所述，要持续关注集成的效果和集成中模型的效果。需要检查模型的结果，找出**每个模型的优势所在**，还要寻找影响集成效果的相对不明显的因素，其中最明显的是模型预测的相关性。通常表现较好但不相关的模型的集成效果更佳。

例如可以用相关度度量和主成分分析来度量数据集内变量信息。同样，可以对每个模型的预测输出应用皮尔逊相关系数（Pearson's correlation coefficient），以理解效果和模型之间的相关性。

回到堆叠集成，集成的模型会输出元特征，这些元特征将用作下一层模型的输入。正如我们会审查更常见的神经网络所使用的特征，需要确保集成中的模型输出的特征可以很好地用作数据集。可以对模型的输出计算皮尔逊相关系数，并将结果用于模型选择。

处理单模型问题时，往往需要花些时间检查问题，并确定合适的学习算法。如果要处理的是一个特征数适中（数十个）的二类分类问题，就需要视情况选择逻辑回归、支持向量机或其他算法。不同方法适用于不同问题，通过试错、并行测试和体验（包括个人体验和线上发布），就可以根据特定输入数据明确实现特定目标的适用方法。

相似的逻辑也适用于集成构建。难点不在于明确单个适合的模型，而在于如何确定能够高效描述输入数据集的不同元素的模型组合，以实现对数据集的整体性描述。了解组件模型的优缺点，并对数据集进行探索和可视化后，就能知道如何通过多次迭代来高效地进行集成了。

最后，在本阶段，数据科学领域存在许多可用的技术。从业者可以应用掌握的的算法知识，在反复迭代中设计出有效的解决方案。

这些解决方案涉及算法知识、模型组合间的交互、模型参数的调整、数据集的转化和集成的处理。不受约束的创造性思维方式同样重要。

Kaggle 的杰出参赛者 Alexander 的作品是很好的例子。可以研究一个具体案例：**精油产品分**

类大赛，了解富有智慧和创造力的数据科学家会选用的一系列方法。

绝大多数模型开发始于这样一个阶段：将不同的解应用于问题，尝试找出数据背后的蹊跷，并弄明白哪种奏效。虽然将 XGboost 视作独立的集成，但本例将它用作堆叠集成的一部分，以生成一些供最终模型使用的元特征。此外，还使用了神经网络和梯度提升树，因为这两种算法都能产生不错的结果。

为了给混合增加一些差异性，Alexander 加入了一个 *K* 近邻模型，因为 *K* 近邻生成的结果（以及元参数）往往与已经包含的模型显著不同。选择输出容易出现差异的组件，是构造高效堆叠集成（以及绝大部分集成）的关键。

为了提升该模型，Alexander 在模型的第二层加入了一些定制的特征元素。在组合 XGboost 和神经网络预测结果时，他还在该层加入了 bagging。至此，该模型用到了本章讲过的大部分技术。除了模型开发外，一些特征工程（比如用 TF-IDF 分析一半的训练数据和测试数据）和识别类差异的绘图技术也贯穿其中。

结合本书介绍的技术，并透彻理解基础算法以及这些技术的搭配效果，在此基础上才能构建出能解决重大数据科学问题的成熟模型。

至此，本书教授了许多基础知识，即实践性知识的基础，这些都是从业者必须掌握的。本书还通过许多现实案例来展现如何用基础知识高效地解决难题。

对于数据科学家，首先要应用这些技术，了解它们效果怎样、有何用途，然后培养高层次的创造性、试验性的思维方式。

8.2　在动态应用中使用模型

前面探讨了如何在相对理想的条件下用技术控制模型效果。具体而言，前提条件是数据可得，且所有数据都可用于模型训练。这些假设在研究或处理一次性问题时通常非常有效，但很多时候不太安全。除了单纯的数据不可用外，还有其他隐患。比如数据科学大赛就需要保留数据集来建立最终的排行榜。

回到前面的话题，回顾一下赢得 Netflix 大奖的 Pragmatic Chaos 算法。当 Netflix 开始评估实现算法时，业务背景和要求都已发生巨变，这使得算法带来的微弱的精度提升（带来的收益）已经比不上其实现成本了。价值 100 万美元的算法已经过时，不会再用于生产。这个案例告诉我们，对商业而言至关重要的是，模型的适应性应尽可能强。

对于机器学习而言，极具挑战性的应用是那些数据随时间（或其他维度）变化的应用。在这种情况下，现有的运行方法将价值减损，而且对于数据发生的许多变化，现有的模型无法通过简单地训练来适应。此时，需要诉诸新技术和新信息。

为了适应和收集这类信息，需要提升预测能力，了解数据可能的变化方式。基于这些信息，模型构建和集成内容就能做出改变，以适应最有可能发生的数据变化。这种适应有助于我们预见性地处理数据变化，缩短调整时间。稍后将看到，在实际应用中，无论幅度大小，根据数据变化进行调整所带来的整体调整时间缩短是有价值的。

下面将介绍一些工具，可增强模型对数据改变的稳健性，还将探讨一些方法。应用这些方法，能在不降低模型性能的前提下，有多个模型可选用，同时适应一种或多种数据变化情况。

8.2.1　理解模型稳健性

准确理解问题是什么、如何解决，以及何时出现非常重要。这包含了对两件事的定义：一是应用于机器学习算法的稳健性，二是数据变化。下面所讲内容属于入门水平，但经验丰富的数据科学家仍能从中挖掘出一些价值。

在专业术语中，机器学习算法的稳健性指的是，算法在应用于训练数据集外的数据集时的水平。

在任何领域，稳健性检验都是机器学习方法论的核心部分。因为机器学习算法易受数据变化的影响，所以验证技术非常重要，即便是最简单的数据开发模型，也要考虑 k 重交叉验证和测试集的使用。

绝大多数的数据集包含信号和噪声。噪声有时可预测（这样更容易控制），有时却随机而难处理。数据集或多或少都会包含噪声。通常包含或多或少的可预测噪声的数据集会比删除过这些噪声的相同数据集更难训练和测试（这点很容易测试）。

在给定数据集上训练模型后，该模型会不可避免地同时基于信号和噪声进行学习。过度拟合的概念通常用于描述模型对给定数据集拟合过好，从而学会了同时基于信号和噪声进行预测，这使得模型对测试集的拟合效果不如对训练数据拟合不那么精确的模型。

训练模型的目的包括尽可能降低任何局部噪声对学习的影响。保留一些数据以便进行测试的验证技术，是为了确保训练过程中对噪声的学习只发生在噪声是训练数据的局部噪声时。可以通过训练误差和测试误差的区别来理解模型的过度拟合程度。

第 1 章应用过交叉验证。检验模型过度拟合情况的另一种有效方法是，以**抖动**（jitter）的方式向训练数据集直接添加随机噪声。2015 年 10 月，Alexander Minushkin 在一篇 Kaggle 笔记中阐述了这项技术，并提到了一个非常有趣的实验。实验的概念很简单：通过添加抖动并观测训练数据的预测精度，可以区分过度拟合模型（添加抖动后，模型的训练误差会增加得更快）和拟合较好或较差的模型（见图 8-4）。

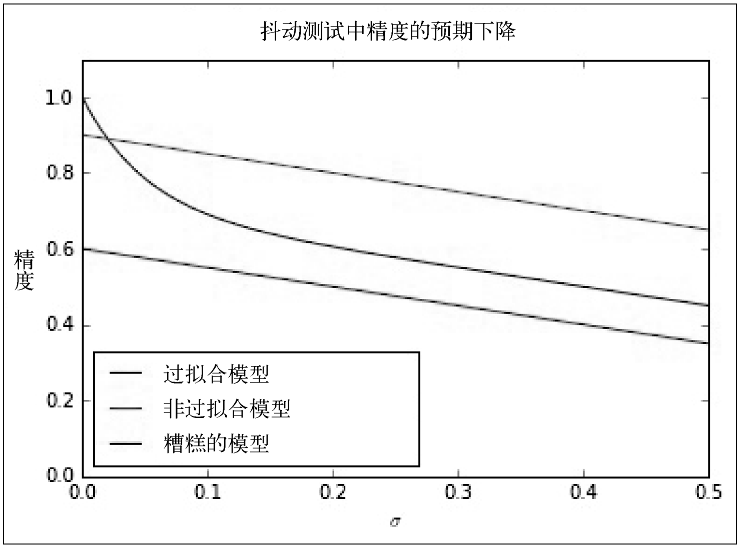


图　8-4

通过绘制抖动测试的结果，可以轻松地判断模型是否出现了过度拟合。过度拟合模型的初始情况非常理想，但其性能往往会随着小抖动的加入而迅速下降。对拟合更好的模型而言，添加抖动带来的性能下降会大幅度削弱。模型内的过度拟合程度在加入较少抖动时尤其明显（而拟合较好的模型的性能胜过过度拟合模型）。

下面介绍如何实现过度拟合的抖动测试。将评分 accuracy_score（指预测正确的类标签占比）作为测试评分的基础。抖动通过（用 np.random.normal）单纯对数据添加随机噪声来定义，噪声的数量则由 scale 参数的配置确定。

```
from sklearn.metrics import accuracy_score

def jitter(X, scale):
    if scale > 0:
        return X + np.random.normal(0, scale, X.shape)
    return X

def jitter_test(classifier, X, y, metric_FUNC = accuracy_score,
                sigmas = np.linspace(0, 0.5, 30), averaging_N = 5):
    out = []

    for s in sigmas:
        averageAccuracy = 0.0
        for x in range(averaging_N):
            averageAccuracy += metric_FUNC( y, classifier.predict(jitter(X, s)))

        out.append( averageAccuracy/averaging_N)

    return (out, sigmas, np.trapz(out, sigmas))

allJT = {}
```

给定分类器、训练数据和一组目标标签的情况下，jitter_test 本身定义为了常规 sklearn 分类的包装器。然后调用分类器，基于经过 jitter 运算的数据进行预测。

下面开始构造大量数据集来进行抖动测试，将用到 sklearn 的 make_moons 数据集。该数据集常用于对聚类和分类算法的性能进行可视化，由两类组成，其数据点能够创建交错的半圆。对 make_moons 数据集加入不同数量的噪声，并取用不同数量的样本后，就可以构造出一系列示例案例来进行抖动测试了。

```
import sklearn
import sklearn.datasets

import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

Xs = []
ys = []
# 少量噪声，大量样本，比较简单
X0, y0 = sklearn.datasets.make_moons(n_samples=1000, noise=.05)
Xs.append(X0)
ys.append(y0)

# 更多噪声，大量样本
X1, y1 = sklearn.datasets.make_moons(n_samples=1000, noise=.3)
Xs.append(X1)
ys.append(y1)

# 少量噪声，些许样本
X2, y2 = sklearn.datasets.make_moons(n_samples=200, noise=.05)
Xs.append(X2)
ys.append(y2)

# 更多噪声，些许样本，比较困难
X3, y3 = sklearn.datasets.make_moons(n_samples=200, noise=.3)
Xs.append(X3)
ys.append(y3)
```

完成这一步后，接下来要创建一个 plotter 对象，用于展现基于输入数据直接得到的模型效果。

```
def plotter(model, X, Y, ax, npts=5000):

    xs = []
    ys = []
    cs = []
    for _ in range(npts):
        x0spr = max(X[:,0])-min(X[:,0])
        x1spr = max(X[:,1])-min(X[:,1])
        x = np.random.rand()*x0spr + min(X[:,0])
        y = np.random.rand()*x1spr + min(X[:,1])
        xs.append(x)
        ys.append(y)
```

```
        cs.append(model.predict([x,y]))
    ax.scatter(xs,ys,c=list(map(lambda x:'lightgrey' if x==0 else 'black', cs)),
                               alpha=.35)
    ax.hold(True)
    ax.scatter(X[:,0],X[:,1],
               c=list(map(lambda x:'r' if x else 'lime',Y)),
               linewidth=0,s=25,alpha=1)
    ax.set_xlim([min(X[:,0]), max(X[:,0])])
    ax.set_ylim([min(X[:,1]), max(X[:,1])])
    return
```

将支持向量机分类器用作抖动测试的基础模型。

```
import sklearn.svm
classifier = sklearn.svm.SVC()

allJT[str(classifier)] = list()

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(11,13))
i=0
for X,y in zip(Xs,ys):
    classifier.fit(X,y)
    plotter(classifier,X,y,ax=axes[i//2,i%2])
    allJT[str(classifier)].append (jitter_test(classifier, X, y))
    i += 1
plt.show()
```

抖动测试可高效评估模型过度拟合程度，与交叉验证相当。Minushkin 通过实践证明了该方法在度量模型拟合质量时比交叉验证有效。

这两种减轻过度拟合的工具适用于以下情况：算法在一次性的数据上运行，或者在基本趋势不会大幅度改变的数据上运行。这对大部分单一数据集问题（比如大部分学术资源库数据集或网络资源库数据集）或基本趋势改变缓慢的数据问题都很适用。

然而，在其他很多情况下，建模中的数据会随时间在一个或多个维度上发生变化。原因可能是捕获数据的方法发生了变化，而这通常是因为新手段或新技术的投入使用。比如自 2005 年起，由于视频录制设备的普及，视频数据的分辨率得到了大幅提升，其质量（和大小）也提高了。无论是否将视频帧本身或文件大小作为参数，都可以看到这类数据的特性、质量和特征分布发生了明显转变。

数据集变量的变化也可能是基本趋势的区别导致的。测度和维度的经典数据模式又可以派上用场了，有助于分析哪些维度影响了测度，从而更好地了解数据变化所受影响的幅度。

时间就是典型例子。许多变量都受制于星期数、月份或季节性变化，具体视情境而定。在许多情况下，一个有效的方法是将这些变量参数化，（前一章探讨过，独热编码等技术有助于算法学习解析这样的趋势），尤其在处理易于预测（比如月份对某地围巾销量的影响）且易于建模的周期性趋势时。

问题更大的时间序列趋势是非周期性变化。正如在某个摄像机案例中，某些时间序列趋势会发生不可逆的变化，而且可能无法预测。软件的遥测容易受到遥测时软件构建的质量和功能的影响。当构建随时间发生变化时，遥测发送的值以及根据这些值构建的变量也会难以预测，陡然改变。

人类行为是许多数据集中的重要因素，可能导致周期性的和非周期性的数据变化。人们在季节性假期前后往往购物更多，但也会基于新的社会或技术发展永久改变购物习惯。

这里增加的一些复杂性不仅来自单个变量及其分布受时间序列趋势影响的事实，还来自相关因素及其相关变量间的交互影响。变量的关系可能会发生可量化的变化。比如人类的身高和体重的关系因时间和地域的不同而不同。可以用 BMI 特征来跟踪这种关系，在不同时间段或不同地域采样，以获知分布的差异。

此外，变量还会发生另一种变化：变量对建模算法的重要性可能会随时间改变。变量的值在一段时间内高度相关，但在另一段时间内相关性较弱。例如气候和天气变量对农业市场的影响。对于一些作物和交易这些作物的企业而言，这些变量在一年的大部分时间内无足轻重，但在作物生长和丰收的季节，这些变量就会变得至关重要。使之更复杂的是，这些因素的重要性也与地域（以及当地气候）有关。

至此，建模的挑战很明确了。对于那些曾经训练过，然后在新数据上运行的模型而言，控制数据变化伴随着严峻挑战。对于那些基于新的输入数据动态重新计算的模型而言，数据变化同样会引发问题，因为变量分布和变量关系会变化，可用的变量对生成有效解的价值也会提升或降低。

成功控制机器学习应用中的数据变化的关键是，识别变化发生时容易影响特征分布、关系和特征重要性的维度（以及引发问题的常见原因），良好的模型应该考虑到这些。

找到容易影响过度拟合的因素，就能更好地开发可以高效控制这些因素的解决方案了。

话虽如此，但构造能解决任何潜在问题的单个模型似乎仍困难重重。简而言之，如果出现了严重的数据变化问题，或许需要尝试其他模型。下面将用集成方法提供更好的解。

识别建模中的风险因素

虽然在许多情况下，随着时间的推移，确定哪些元素对模型构成风险很简单，但通过结构化的过程进行识别也可能会有帮助。下面简要介绍一些思路和技术，可用于保护模型免受数据变化风险的影响。

大多数数据科学家会保留数据集的一份数据字典，以用于一些常见场景或自动化应用。数据或应用往往很复杂，而保留数据字典通常是很好的做法。在识别风险因素时，浏览这些特征，然后基于不同的风险类型为它们添加标签，是高效的做法。

常用的标签如下所示。

- **纵向可变**：该参数在长期内是否容易发生变化？这个疑问的产生往往是因为现有数据集的范围无法完全展现数据的纵向趋势。四季就是典型的例子，它能影响许多地区的人类行为，还能影响一些与基础气候变量密切相关的事物。其他纵向趋势包括财政年度和工作月，还可以扩展至与地区相关的其他许多纵向趋势。新 iPhone 模型的生命周期或野鼠的种群流动可能也是重要的纵向因素，这取决于研究性质。
- **缓慢变化**：该类型参数是否会随时间而添加新值？这个概念借鉴了数据仓库的最佳实践。传统意义上，缓慢变化的维度会增加新的参数编码（比如随着一家新店的开业或一个新案例的确定）。如果没有得到恰当的处理或新参数过多，那么它们会彻底颠覆模型。缓慢变化数据的另一种影响更难处理，它会影响特征的分布，从而严重影响模型的性能。
- **关键参数**：将数据值监控和决策边界/回归方程的重新计算相结合，往往能比较好地处理一些缓慢变化的数据和季节性变化。但如果大量未预料到的新案例或案例类型出现，尤其是当它们影响的变量严重依赖模型时，需要采取相应行动。因此，还要清楚哪些变量最依赖解决方案。

这样添加标签的过程非常有帮助（不仅是作为记忆的输出），主要是因为它有助于执行以下操作。

- 明确预期并制定一份清单，以便监控准备工作。如果连记录纵向可变和参数的缓慢改变都无法做到，除了重新计算时模型所侧重的参数的变化及其（可能缓慢下降的）性能测度，基本上无法了解模型的任何输出了。
- 研究缓解方案（比如改进标准化方法或增加参数，这些参数将基于数据维度的变化进行编码）。缓解方案和参数的添加许多时候是处理数据变化的最佳解决方案。
- 用构造好的数据集进行稳健性测试，数据集中的风险特征要有意地做出改动，以模拟数据变化。在这些条件下，对模型进行应激试验，找出模型能容纳的变化幅度是多少。得到这项信息后，就能轻松而有把握地利用试验结果来构建早期预警系统了。当数据的变化超过特定的安全阈值后，就能预估模型性能的下降幅度了。

8.2.2 控制模型稳健性的策略

前面介绍了大量有效的集成技术，这些技术有助于模型在性能和稳健性之间达到平衡。然而在使用技术时，需要决定如何以及何时降低模型性能来增强稳健性。

的确，贯穿本章的一个主题是，如何平衡自相冲突的目标，即构造一个高效、高性能的模型，同时该模型又不会太过呆板而无法响应数据变化。前面介绍的许多解决方案都是牺牲一面来成全另一面，这并不太理想。

此时需要稍微放宽选择，介绍一些值得称赞的技术。发展中的商业格局对稳健、高效的统计

模型的需求由来已久，探索与实现颇多。信用风险建模等领域有着悠久的应用统计建模历史，并且为了解决需求，发展出了有效的决策管理方法。数据科学家可以用一些现有技术来构建模型，为自己所用。

其中一种高效方法是**冠军/挑战者测试**（即 A/B 测试），这是一种以测试为中心的方法，涉及多个并行的模型配置。除了应用训练的模型输出（比如指导商业活动或支持数据报告）外，冠军/挑战者测试还会训练一个或多个备选模型配置。

通过维持和监测多个模型，当替代模型表现更好时，可以用其替换掉正在使用的模型。这通常通过计算所有模型的性能评分并观察结果来完成，因此可以人为决定是否以及何时切换到挑战者模型。

虽然最简单的实现可能是，挑战者模型效果超过主模型时立即切换到挑战者模型，但这种做法其实很少见，因为有的挑战者模型存在陷入局部最小值的风险（比如一周中的某一天或一年中的某一月的局部趋势）。花大量时间评估挑战者模型是常事，尤其当应用比较敏感时。在复杂的实际情况中，甚至可能需要做额外的测试，即给有潜力的挑战者模型提供案例样本，判断挑战者实现的显著提升能否超过冠军。

除了简单的“取代挑战者”接替规则外，还有一些创新的空间。基于投票的方法很常用，训练后的集成中排名靠前的子集逐个提供分数，这些分数就作为（加权或不加权的）投票。另一种方法需要使用**波达计数法**（Borda count）。这是一种投票系统，每个投票者按照喜好给参赛的解决方案排序。在集成的情况下，往往会给每个单独模型的预测分配一个点值，该点值等于其逆排名（保持每个模型独立），然后将这些投票组合起来（通常要尝试一系列不同的权重），生成一个结果。

在使用大量模型时，投票可以表现得相当好，但这取决于具体的建模环境及不同投票者间的相似性等因素。如前所述，关键是要使用皮尔逊相关系数这种测试，来确保模型集表现良好且不相关。

有时输入数据的特定类（比如附带具体分类标签的用户）用给定的挑战者处理会更加高效，而且可以实现一个路由系统，其中多个冠军处理不同的用户子群。这种方法和提升集成法的优势有些重合，但能通过区分关注点的方式在生产环境中发挥作用。然而，维护多个冠军会增加数据团队监测和管理的负担，因此，除非必要，最好避免使用该方法。

需要考虑的主要问题是，如何对模型评分，这也是迫切的现实挑战。尤其是很难在现实情境中比较多个模型，因为通常没有类标签（用于指导正确性）。在预测情境中，通常会用冠军模型的预测指导行动，这会影响预测事件，使得该问题更加复杂，难以判断挑战者模型的预测效果如何。如果基于冠军模型的预测采取行动，就无法确定模型的结果。

常规的实现过程是，给每个挑战者模型提供一个可统计的输入数据样本，然后比较两种方法

的提升效果。这种方法本质上会限制用于帮助解决建模问题的挑战者的数量。另一种选择是在数据处理的过程中特意保留一个可统计的样本，并用该样本构造一个回归检验。对所有冠军模型和挑战者模型进行检验，可以为比较提供一个有意义的基础。

该方法的弊端是，无论为测试用例生成正确的类标签需要多长时间，高效模型始终需要随数据变动而更改。虽然在很多情况下这个弊端并不致命（冠军模型在生成精确模型期间仍然起效），但相对于模型训练时间，数据的基本条件改变过快仍会引发很多问题。

> 这里简单解释一下模型训练时间和数据变化频率的关系。应用机器学习的典型目标是，面对数据变化频率的影响，尽可能缩短训练时间。最坏的情况是，如果训练模型的时间长于提升模型精度所需要的时间（比率≥1），那么模型就永远不会生成可直接指导当前行动的结果。总体而言，高比率表示需要对模型或数据进行一些审查和调整（审查在较低的置信度下更快的分数传递是否传递了更多价值，或者调整环境变量可控部分的变化速度）。
>
> 比率变得越小，团队就越有可能用模型输出来指导行动并产生价值。这个比率有益于设计自动化建模解决方案，可以在组织内推广，具体取决于其在建模环境中的多变性和可量化性。

备选模型可以是下一个表现最佳的集成配置，也可以是留待观察的旧模型。在复杂的操作中，会配置一些挑战者来处理不同的假设情景（比如**如果该地区的温度低于期望值 2 摄氏度，或如果销售明显低于预期**）。这些模型使用训练主模型的数据进行训练，或者用模拟情景的有偏向或准备好的数据进行训练。

如果挑战者并不全是功能相同的微小变体，那么越多越好（能提升模型稳健性和性能）。挑战者模型还能为创新和测试提供安全场所。通过观察高效的挑战者，可以有效地洞察在一系列可能的环境变化中冠军集成的稳健度。

这些技术有助于应对环境的不断变化，将现有模型应用于实际场景。本章还探讨了将机器学习模型应用于生产时会出现的复杂情况。比如样本间或维度间的数据变化可能导致模型性能逐渐降低。掌握了数据变化的概念后，就能更好地识别风险并预知它将在何处以及如何出现。

下面将介绍提升模型稳健性的技术。前面讲过如何观察底层数据来识别模型退化风险，并介绍了一些有用的启发式方法。我们借鉴了现有的决策管理方法，并采用了一种广受好评、在应用机器学习等领域应用已久的手段，即冠军/挑战者测试。该过程有助于在比赛中组织并测试多个模型。具有前瞻性的模型替代战略和高效的性能检测有助于实现对模型生命周期和质量更方便可控的管理，同时可提供许多有价值的运营见解。

8.3　扩展阅读

关于集成和集成类型的全面介绍，参见 Kaggle 参赛者 Triskelion 的文章（http://mlwave.com/kaggle-ensembling-guide/）。

有关赢得 Netflix 大奖的 Pragmatic Chaos 的介绍，参见 http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BellKor.pdf。Netflix 关于业务背景的变化如何使这个价值 100 万美元的模型失效的解释，可参考 Netflix Tech 的博文（http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html）。

要想了解如何将随机森林集成应用于商业环境，以及各种重要的诊断表和推理过程，参见 Arshavir Blackwell 的博文（https://citizennet.com/blog/2012/11/10/random-forests-ensembles-and-performance-metrics/）。

有关随机森林的更多信息，可参考 scikit-learn 的文档（http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html）。

XGboost 文档（https://xgboost.readthedocs.io/en/latest/tutorials/model.html）详细介绍了梯度提升树。

有关 Alexander Guschin 参加精油产品分类大赛的专稿，参见 No Free Hunch 的博文（http://blog.kaggle.com/2015/06/09/otto-product-classification-winners-interview-2nd-place-alexander-guschin/）。

关于 Alexander Minushkin 对过度拟合所做的抖动测试，参见 https://www.kaggle.com/miniushkin/introducing-kaggle-scripts/jitter-test-for-overfitting-notebook。

8.4　小结

本章内容丰富。首先介绍了集成，这是机器学习大赛中最强大且常用的技术之一，然后将专业知识和实际案例相结合，进一步介绍了将集成应用于机器学习项目所涉及的理论和代码。

此外，本章还专门探讨了将模型一次运行数周、数月时会产生的相关问题。我们讨论了数据变化的含义、识别手段，以及预防方法，还介绍了如何构造并行的模型集，可以根据季节性变化或模型集的性能变化对该模型集进行转换。

在讲解这些技术的过程中，花了大量时间研究实际案例，旨在介绍顶尖数据科学家需要具备的创造性思维和各种知识。

基于本书所讲的技术知识、代码复用，以及各种可能性，现在你可以处理实际的数据建模问题了。

第 9 章 其他 Python 机器学习工具

前 8 章介绍并应用了一系列技术，它们有助于在很多应用场境中扩充数据并对数据建模。

前面提到了 Python 库的组合，尤其是 NumPy 和 Theano，具体算法中应用了其他一些库。前面没有深入介绍其他可选工具、这些工具的特点及探讨意义。

本章旨在介绍其他一些可用的关键库和框架。这些工具有助于合理化和简化构造和应用模型的过程。本章将展示这些工具及其应用，9.2 节将给出一些参考资源。

成功解决数据科学问题并成为专业数据科学家的主要因素在于，透彻理解算法和库的最新发展。作为专业人士，数据科学家往往高度依赖所用数据的质量，但使用出色的工具也非常重要。

本章将介绍数据科学家可用的一些有力的工具、它们的优势，以及如何在工作中将它们与前几章介绍的工具和技术搭配使用。

9.1 可选的开发工具

近几年出现了大量能为工作流带来便利的机器学习新框架。它们往往专注于具体的用例或目标。这让它们非常实用，甚至成为了必备工具，但这也意味着需要用到多个用于改良工作流的库。

越来越多的 Python 机器学习新项目力图解决具体的工作流问题，其中两个库已纳入了工作流，它们能加速或改进前几章完成的工作，值得一讲。本章将介绍 Lasagne 和 TensorFlow，探讨两个库中的代码和功能，以及将它们纳入工具集的原因。

9.1.1 Lasagne简介

实践中有时用 Python 构造模型耗时超过预期。但这种做法对复杂的模型非常有效，并且获益颇多（比如 GPU 加速和可配置性）。在处理简单案例时，类似于 Theano 的库用起来会相对复杂。这显然不太好，因为人们往往想使用简单的模型，比如在建立基准时。

Lasagne 是由深度学习和音乐数据挖掘研究团队开发的，用作 Theano 的一个接口。它专门用

于为新模型快速高效地设计原型。

该目标决定了 Lasagne 的构造方式：以一种比原生 Theano 代码编写的运算更简单且更易理解的方式调用 Theano 函数，并返回 Theano 表达式或 `numpy` 数据类型。

下面介绍 Lasagne 背后的概念模型，应用一些 Lasagne 代码，并探讨该库能给当前实践带来的价值。

认识 Lasagne

Lasagne 采用“层”的概念。这是机器学习中的一个常见概念，指一系列神经元和运行规则。这些规则用于获取输入值，生成分数、标签或其他转换结果。通常可将神经网络视作一组层，从层的一端输入数据，另一端输出值（虽然具体的处理方式多种多样）。

在深度学习中，常将单独的层视作“一等公民”。在机器学习工作中，通常基于仅使用少量参数规范（比如节点计数、偏差和权重值）的层建立网络。

近年来，数据科学家开始利用该优势配置单个层。如今，高级机器学习领域包含子模型和转化后输入的层已屡见不鲜。即便是特征，也可能根据需求跳过层，或者中途给模型的层添加新特征。例如谷歌用于解决图像识别问题的 CNN 架构。为了提升性能，这些网络大规模地改进了层。

因此，Lasagne 将层作为基础模型部件来处理是十分合理的。它在模型创建过程中添加的功能将不同的层快速直观地叠加到模型中。可以简单地调用 `lasagne.layers` 中的一个类，将类堆叠在自己的模型上。实现代码非常高效，如下所示。

```
l0 = lasagne.layers.InputLayer(shape=X.shape)

l1 = lasagne.layers.DenseLayer(
    l0, num_units=10, nonlinearity=lasagne.nonlinearities.tanh)

l2 = lasagne.layers.DenseLayer(l1, num_units=N_CLASSES,
                               nonlinearity=lasagne.nonlinearities.softmax)
```

只需要 3 条简单的语句，就可以用简单且可配置的函数创建出网络的基本结构。

这段代码构造了一个 3 层模型。层 `l0` 调用了 `InputLayer` 类，以作为模型的输入层。基于输入的预期形状（通过 `shape` 参数定义），该层将输入数据集转化成 Theano 张量。

接下来的层 `l1` 和层 `l2` 都是全连接层（也称“密集层”）。层 `l2` 定义为输出层，所含单元数等于类数，层 `l1` 则使用了同一个 `DenseLayer` 类来构造一个包含 10 个神经元的隐藏层。

除了 `DenseLayer` 类可用的标准参数（权重、偏差、单元数和非线性类型）的配置外，还可以用不同类部署完全不同的网络。Lasagne 为各种常见的层提供了类，包括密集层、卷积池化层、递归层、标准化层和噪声层等。此外，还有一个用途特殊的层类，它能提供许多额外的功能。

如果需要定制新功能，用户也可以轻松地自定义层类型，并将其与其他 Lasagne 类一起使用。然而，对于大多数原型设计和快速迭代开发环境而言，这需要大量准备工作。

Lasagne 提供了一种类似的简洁界面，用于定义网络的损失计算。

```
true_output = T.ivector('true_output')
objective = lasagne.objectives.Objective(l2, loss_function=lasagne.
                                          objectives.categorical_crossentropy)

loss = objective.get_loss(target=true_output)
```

这里定义的 loss 函数是常用函数之一，其他常用函数还包括平方误差、二类问题和多类问题的铰链损失，以及交叉熵函数。这里还提供了用于验证的精确度评分函数。

有了 loss 函数和网络架构这两个组件，就可以开始训练网络了。为此，需要多写几行代码。

```
all_params = lasagne.layers.get_all_params(l2)
updates = lasagne.updates.sgd(loss, all_params, learning_rate=1)
train = theano.function([l0.input_var, true_output], loss, updates=updates)

get_output = theano.function([l0.input_var], net_output)

for n in xrange(100):
    train(X, y)
```

这段代码利用 theano 实例的内置函数来训练示例网络，并用 loss 函数迭代地进行训练，以对给定输入数据集进行分类。

9.1.2 TensorFlow简介

第 4 章介绍谷歌对 CNN 的使用时，建立了一个基于卷积的多层“性能怪兽”。 为了应对更复杂的挑战，网络在层数和复杂性上不断扩展，如何构造并监控这种网络变得愈发重要。

为了处理该问题，谷歌的机器智能研究团队开发了一个名为 TensorFlow 的库，用于简化复杂的机器学习模型的优化和建模。

TensorFlow 凭两点实现了该目标：一是在常用的结构（比如 NumPy 对象）上提供了简洁的编程接口（这里指 Python API），二是提供了功能强大的故障诊断和图像可视化工具（比如 TensorBoard），大大便利了数据结构优化。

1. 认识 TensorFlow

数据科学家可以用 TensorFlow 将数据转化操作设计成计算图上的一个流。可以扩充和调整这张图，也可以随意调整单独的节点，从而优化单独的层或模型组件细节。TensorFlow 工作流通常包括两个阶段，第一阶段为构建阶段，即图的生成阶段。

在构建阶段中，可以用针对 TensorFlow 的 Python API 编写代码。与 Lasagne 类似，TensorFlow 提供了编写网络层的相对简单的接口，只需要在构造层之前确定权重和偏差。接下来的示例展示了在构造卷积层和简单的最大池化层（每种只用了一行代码）之前，权重和偏差变量的初始设置。此外，还用 `tf.placeholder` 来生成输入数据的占位符变量。

```
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

W = tf.Variable(tf.zeros([5, 5, 1, 32]))
b = tf.Variable(tf.zeros([32]))


h_conv = tf.nn.relu(conv2d(x_image, W) + b)
h_pool = max_pool_2x2(h_conv)
```

可以扩展该结构使之包括一个 `softmax` 输出层，就像对 Lasagne 做的那样。

```
W_out = tf.Variable(tf.zeros([1024,10]))
B_out = tf.Variable(tf.zeros([10]))

y = tf.nn.softmax(tf.matmul(h_conv, W_out) + b_out)
```

与直接在 Theano 和 Python 库中编写相比，可以看到迭代时间明显缩短。与 Python 相比，用 C++编写的 TensorFlow 也提升了性能，在执行时间方面具有优势。

接下来要训练和评估模型。这里定义训练用的 `loss` 函数（此处指交叉熵）、验证用的 `accuracy` 函数，以及优化策略（此处指最陡梯度下降法），代码如下所示。

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
                               reduction_indices=[1]))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y_,1), tf.argmax(y_,1))

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

然后就可以开始迭代运行模型了。整个过程简单直接。

```
sess.run(tf.initialize_all_variables())
for i in range(20000):
  batch = mnist.train.next_batch(50)
  if i%100 == 0:
    train_accuracy = accuracy.eval(feed_dict={
      x:batch[0], y_: batch[1], keep_prob: 1.0})
    print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

2. 用 TensorFlow 迭代改进模型

前面的单个模型也显示了使用 TensorFlow 带来的好处。它能为开发复杂结构和训练方法提供简单的接口，让我们可以更轻松地使用前面介绍的算法。

但如前所述，开发初始模型只是模型开发过程的一小部分。为了提升模型的质量，往往需要反复测试和剖析模型。这通常不局限于用单个库或技术来实现，而且模型间的测试和监控解决方案也并非一成不变。

TensorFlow 可用于在迭代过程中深入了解模型，这便是模型开发的执行阶段。在该阶段中，可以利用 TensorFlow 团队提供的工具来探索并改善模型。

这些工具中最重要的可能是 TensorBoard 了，它可以实现模型的可视化。TensorBoard 有多种功能，包括显示基本模型信息（包括测试和训练每次迭代时的性能测度）的仪表板（见图 9-1）。

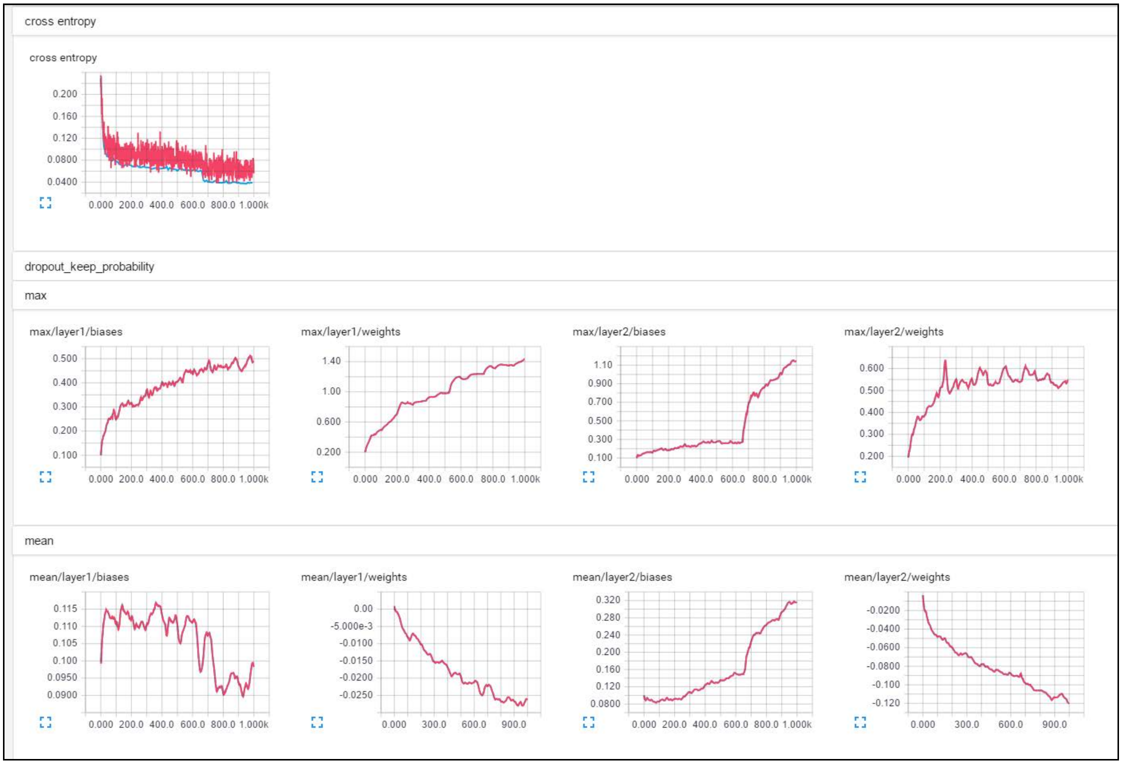


图 9-1

此外，TensorBoard 仪表板还提供了底层信息，其中包括模型每一层的权重、偏差和激活值的变化范围。这是迭代过程中非常有用的诊断信息。这些数据易于获取，且很有价值（见图 9-2）。

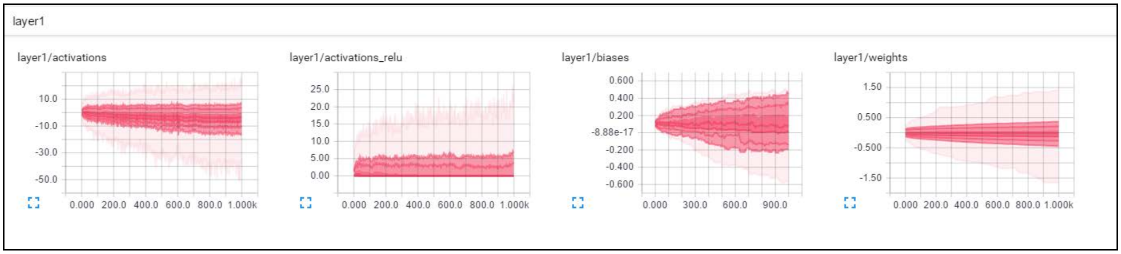


图　9-2

此外，TensorBoard 可生成给定模型的张量流的细节图。张量是 *n* 维数据数组（此处指 *n* 个特征），它与“**输入数据集**”一词关系紧密。应用于张量的一系列运算叫作“张量流”，这是 TensorFlow 中的一个基本概念。优化和调试机器学习模型时，最重要的是了解模型及其运行的信息，即使是很底层的信息（见图 9-3）。

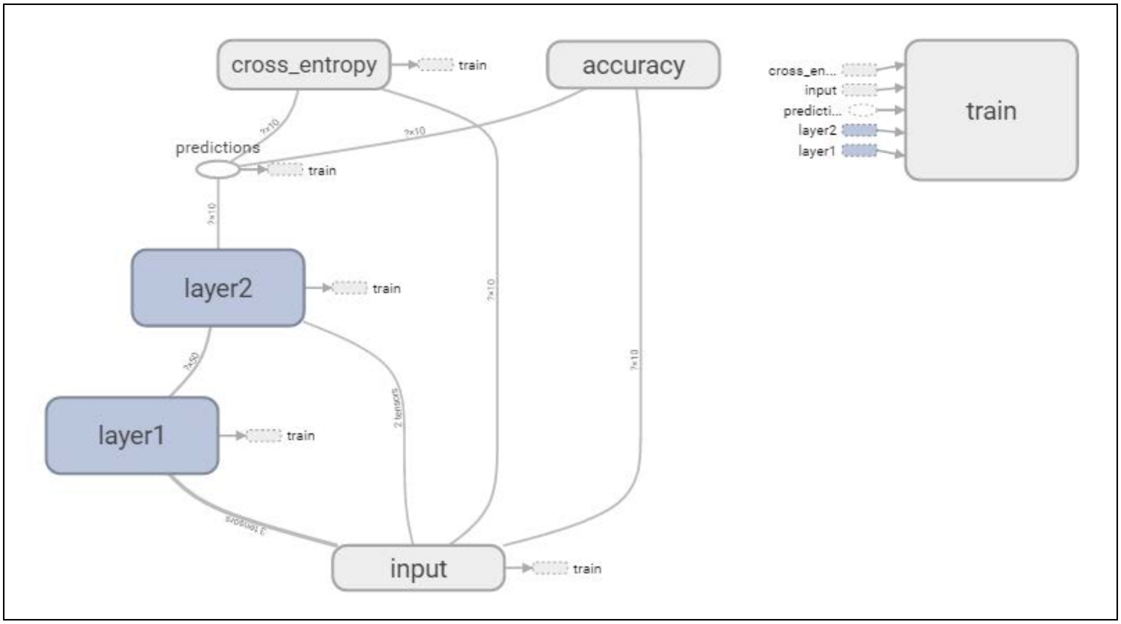


图　9-3

TensorBoard 图像能通过变量细节展示模型的结构。概览之后，可以深入研究模型的各个组件和连续的子元素。本例中，可以看到第二个网络层的 dropout 函数中发生的特定运算。基于所见，可以判断出下次迭代需要调整的地方。

这种透明度不太常见，而且非常有助于调整模型组件，尤其当模型元素或层表现不佳时（比如 TensorBoard 图像可以显示层的元参数值或网络的整体表现）。

可以通过事件日志创建 TensorBoard，也可以在 TensorFlow 运行时生成 TensorBoard。在日常使用 TensorFlow 的开发过程中，TensorBoard 很容易生成和使用（见图 9-4）。

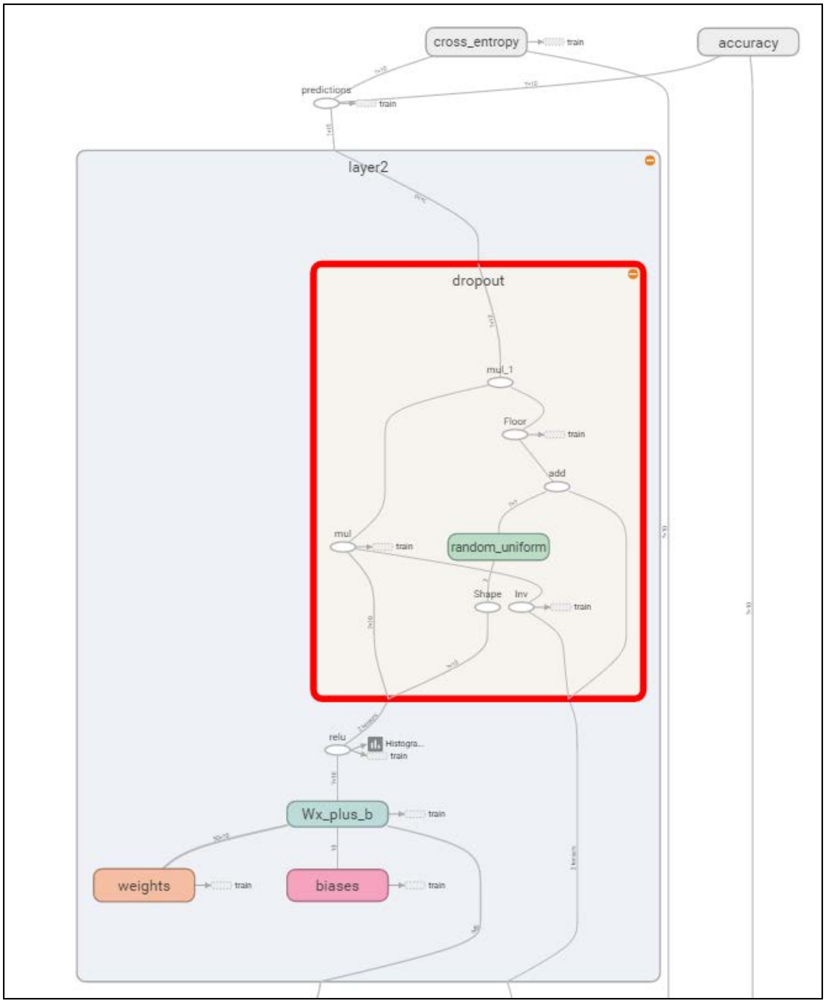


图　9-4

2016 年 4 月底，DeepMind 团队加入了谷歌大脑团队，和许多研究人员及开发人员一起使用 TensorFlow。谷歌已将 TensorFlow 开源，并承诺将继续支持 TensorFlow，助力模型开发和改进。

9.1.3　何时使用这些库

阅读本书时，读者可能会有疑问：“**既然如此，为什么不直接在开头就介绍这个库呢？**”当本章介绍一些可简化工作的优良接口时，读者可能会不由地提问：“为什么要花时间研究 Theano

函数和其他底层信息呢？”

当然，我提倡使用最好的工具，尤其是对于原型设计工作来说，因为原型设计的价值更多在于理解大概的适用范围或识别特定的问题类型。下面说明本章才介绍这些库的三个原因。

第一个原因是这些工具的作用有限。它们确有不少功能，根据领域和领域中问题的性质，有的数据科学家能用它们解决大部分深度学习需求。但超过特定性能水平和问题复杂程度后，就需要理解如何用 Theano 构造模型、从零开始构造评分函数，或利用本书提到的其他技术。

着重底层实现的另一个原因与涉及的技术逐渐成熟有关。如今 Lasagne 和 TensorFlow 当然值得探讨和推荐。此外，很多项目（本章未提及的一些杰出框架，比如 Keras、Blocks 和 Pylearn2）是基于 Theano 的。

即使是现在，其他库和工具在一两年内变成讨论话题或默认工作环境也是完全有可能的。该领域发展极快，主要是因为关键企业和研究团队的深耕。他们一直在创建新工具，因为旧工具达到了使用极限，或者他们找到了改进工作的方法。

注重底层的另一个原因是，本书与之相关。本书将理论和代码相结合，用代码来教授理论。将算法的原理抽象化，只讨论如何用它们处理具体问题当然会很吸引人。本章介绍的工具能让从业者在一些问题上取得不错的成绩，而无须理解调用的函数，但我认为这不是培养数据科学家的好方法。

如果想处理微妙而难解的数据问题，需要能修改和自定义算法，还需要懂得如何选择合适的解决方案，而本书给出了一些相关细节。此外，本书因篇幅所限未能提供一些更具体的信息。如果掌握了这些细节，就能灵活而透彻地应用深度学习算法。

清楚哪些工具能完成任务，而哪些不能同样重要。目前，Lasagne 非常适合这样一种情况：为了建立基准或早期测试而开发新模型，在这种情况下需要优先考虑迭代速度和获取结果。

此外，TensorFlow 适合模型开发周期的后期。当收益不再像前期那样易得，需要花费大量时间调试和改进模型时，TensorFlow 相对快速的迭代就成了明显优势，但这其实是 TensorBoard 的诊断工具带来了压倒性的附加价值。

因此，应将这两个库加入工具集。根据当前问题的性质，这些库以及其他库将会派上用场。

9.2　扩展阅读

Lasagne 用户指南非常全面，值得一读。

也可以查看 TensorFlow 的使用说明。

9.3 小结

最后一章，是对之前有关算法、配置和诊断的探讨的拓展，介绍了在实现深度学习算法时能提升体验的工具。

本章介绍了使用 Lasagne 的好处。作为 Theano 的接口，Lasagne 能加速和简化模型的早期原型设计。本章还研究了 TensorFlow，这个由谷歌开发的库可用于辅助深度学习模型的调整和优化。它能轻松实现模型性能的清晰可视化，且有助于简化复杂模型的诊断和调试。

这两种工具在我们的工作流程中都有一席之地，适用于不同的问题。

本书介绍了各种高级机器学习技术，涵盖了基本算法和概念，以及如何运用强大而流行的工具集。

除了这些技术，本书还想传达一个更深刻的概念，这个基础概念很难教授和学习，同时也是实现机器学习最佳性能的关键。

机器学习领域发展极快。这种速度体现在几乎每周学术期刊或行业白皮书上都会发布新得分。例如对 MNIST 这样的案例的训练迅速从意义重大的挑战变成了"**儿戏**"（鸢尾花数据集的深度学习版）。此外，该领域还迎来了新的重大挑战：CIFAR-10、CIFAR-100。

与此同时，该领域的发展出现了循环。Yann LeCun 等学者在 20 世纪 80 年代提出的概念逐渐复苏，因为计算结构和资源增长使得大批量的实际数据更方便使用。为了最大限度利用一些新技术，必须理解那些数十年前定义的概念，而它们又是基于更久之前定义的其他概念而定义的。

本书旨在平衡各方。掌握前沿动态和先进技术非常重要，理解新定义的技术概念以及近几年所做的调整同样重要。

最重要的是，本书展示了这些体系结构和方法的可塑性。在数据科学实践中，思路往往殊途同归：针对特定问题的最佳解决方案也是特定的。

这就是 Kaggle 大赛获胜者进行大量特征准备和结构调整的原因，也是谷歌开发 TensorFlow 来显示模型架构的细微属性的原因。能流畅地调整、实现并组合算法的理论和应用时，就意味着真正掌握了机器学习技术。

纵观本书提及的诸多技术和案例，希望能将对数据问题的思考，以及对算法的操作和配置尽皆相授，助你成为数据科学家。本书"**扩展阅读**"中推荐的许多案例意在扩充知识，助你进一步探究本书所授技能。

最后，祝你在模型构建和算法配置过程中一切顺利，也希望你能享受其中，从中获益。

附 录

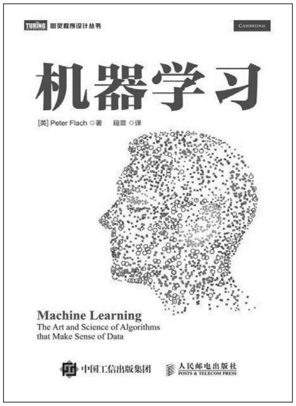
代码运行要求

本书内容利用了公开可用的数据和代码，包含开源的 Python 库和框架。每章的示例代码都附带一个 README 文件，其中记录了运行该章脚本中提供的代码所需要的所有库。方便起见，我们将这些文件的内容整理如下。

需要注意的是，前几章要求安装的库，后续几章的代码也可能会用到。这些库可以通过关键词识别。尤其重要的是，要设置好第 1 章中提及的库，以便后续章节使用。每一章的要求如下表所示。

章　　号	要　　求
1	• Python 3（推荐 3.4） • sklearn（NumPy, SciPy） • matplotlib
2~4	• theano
5	• Semisup-learn
6	• Natural Language Toolkit（NLTK） • BeautifulSoup
7	• Twitter API 账户
8	• XGBoost
9	• Lasagne • TensorFlow

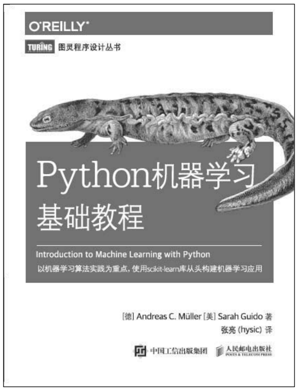
技术改变世界 · 阅读塑造人生



机器学习

◆ 被誉为内容最全面的机器学习教材，Machine Learning期刊主编力作

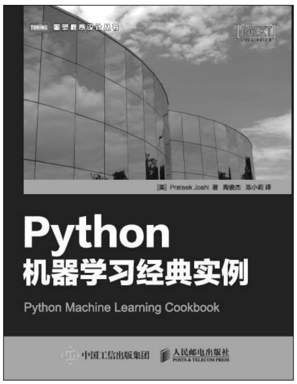
作者： Peter Flach
译者： 段菲



Python 机器学习基础教程

◆ 以机器学习算法实践为重点，使用scikit-learn库从头构建机器学习应用

作者： Andreas C. Müller，Sarah Guido
译者： 张亮



Python 机器学习经典实例

◆ 监督学习技术、预测建模、无监督学习算法等前沿话题的实例代码展示
◆ 来自Kaggle的经典数据集和机器学习案例
◆ 用流行的Python库scikit-learn解决机器学习问题

作者： Prateek Joshi
译者： 陶俊杰，陈小莉

近些年机器学习发展迅猛，早已从科研领域延伸到了商业领域，新技术和新应用不断涌现，其中Python语言光芒难掩。越来越多的开发者和研究人员选择了Python，Python也助推了机器学习和数据科学的发展。

本书是基于Python的机器学习指南，全面介绍了各种强大的机器学习技术，不仅清晰阐释了基本的算法和概念，而且运用丰富的Python代码示例展示了各种复杂算法的实现与应用，此外还介绍了如何使用流行的机器学习库和框架。对于想要了解数据科学领域中的新进展，以提高自身技能的Python开发人员、分析师和数据科学家来说，本书值得一读。

- 利用无监督学习技术识别数据集的内在模式与结构
- 掌握受限玻尔兹曼机和深度信念网络的工作原理
- 使用堆叠降噪自编码机构建深度架构
- 应用卷积神经网络解决图像分类问题
- 学习CPLE、自学习和S3VM等半监督学习技术
- 运用强大的特征工程技术提升模型性能
- 使用集成技术提升深度学习模型的效率
- 介绍Lasagne、TensorFlow等机器学习工具

Packt>

图灵社区：iTuring.cn

热线：(010)51095183转600

分类建议　计算机/机器学习/Python

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-52968-8

ISBN 978-7-115-52968-8

定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱: ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks